USENIX conference proceedings

LISA XV Proceedings of the 15th Systems Administration Conference San Diego, California, Dec. 2–7, 2001

LISA XV

Proceedings of the Fifteenth Systems Administration Conference

San Diego, California, USA
December 2–7, 2001

Sponsored by **The USENIX Association** and
SAGE, the System Administrators Guild

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

SAGE
THE SYSTEM ADMINISTRATORS GUILD

For additional copies of these proceedings contact

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Telephone: 510-528-8649
http://www.usenix.org
<office@usenix.org>

The price is $35 for members and $45 for nonmembers.

Past USENIX Large Installation Systems Administration Workshop and Conference Proceedings (price: member/nonmember)

Systems Administration VI Conference	1992	Long Beach, CA	$23/$30
Systems Administration VII Conference	1993	Monterey, CA	$25/$33
Systems Administration VIII Conference	1994	San Diego, CA	$22/$29
Systems Administration IX Conference	1995	Monterey, CA	$30/$38
Systems Administration X Conference	1996	Chicago, IL	$30/$38
Systems Administration XI Conference	1997	San Diego, CA	$30/$38
Systems Administration XII Conference	1998	Boston, MA	$32/$40
Systems Administration XIII Conference	1999	Seattle, WA	$32/$40
Systems Administration XIV Conference	2000	New Orleans, LA	$35/$45

Outside the U.S.A. and Canada, please add $12 per copy for postage (via air printed matter); $13 for LISA XIV & XV.

ISBN 1-880446-05-7

♻ Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the Fifteenth Systems Administration Conference (LISA XV)

December 2-7, 2001
San Diego, CA, USA

CONTENTS

Monte LISA Overdrive: Empirical System Administration

Wednesday (4:00-5:30 pm) **Chair: William Annis**

Seeing How the LAN Lies: Network Monitoring

Thursday (9:00-10:30 am) **Chair: John Sellens**

Level 1 Diagnostics: Short Topics On Host Management

Thursday (11:00 am-12:30 pm) **Chair: Adam Moskowitz**

To Your Scattered PC's Go!: Distributed Configuration Management

Thursday (2:00-3:30 pm) **Chair: Jon Stearley**

Human Interface: Timely Solutions

Friday (9:00-10:30 am) **Chair: Ozan Yigit**

The Network From Orbit: A Global Perspective

Friday (9:00-10:30 am) **Chair: John Sellens**

Adapting the Collective: Short Topics on Configuration Management

Friday (11:00 am-12:30 am) **Chair: Sigmund Straumsnes**

AUTHOR INDEX

ACKNOWLEDGMENTS

PROGRAM CHAIR
Mark Burgess, *Oslo University College*

PROGRAM COMMITTEE
Eric Anderson, *U. Cal. Berkeley/HP*
William Annis, *University of Wisconsin*
Alva Couch, *Tufts University*
Emmett Hogan, *Certainty Solutions*
Adam Moskowitz, *Menlo Computing*
Phil Scarr, *Certainty Solutions*
John Sellens, *Certainty Solutions*
Jon Stearley, *Compaq Federal*
Sigmund Straumsnes, *Oslo University College*
Ozan S. Yigit, *Sun Microsystems*

INVITED TALKS COORDINATORS
Esther Filderman, *Pitt. Supercomputing Center*
Tom Limoncelli, *Lumeta Corp.*

NETWORK/SECURITY TRACK
Cat Okita, *Earthworks*
Tom Perrine, *San Diego Supercomputer Center*

WORKSHOP COORDINATOR
Pat Wilson, *University of California, San Diego*

AFS WORKSHOP
Esther Filderman, *Pitt. Supercomputing Center*
Derrick Brashear, *Carnegie Mellon University*
Ted McCabe, *MIT*

BEST PRACTICES IN INTRUSION ...
Tom Perrine, *San Diego Supercomputer Center*
Pat Wilson, *University of California at San Diego*

TEACHING SYSADMIN WORKSHOP
John Sechrest, *PEAK, Inc.*
Curt Freeland, *University of Notre Dame*

CFENGINE WORKSHOP
Mark Burgess, *Oslo University College*

METALISA WORKSHOP
Cat Okita, *Earthworks*
Tom Limoncelli, *Lumeta Corp*

TAXONOMY WORKSHOP
Rob Kolstad, *Delos*

ADVANCED TOPICS WORKSHOP
Adam Moskowitz, *LION Biosciences Res.*

WORK-IN-PROGRESS COORDINATOR
Emmett Hogan, *Certainty Solutions*

GURU-IS-IN COORDINATOR
Lee Damon, *University of Washington*

TERMINAL ROOM COORDINATOR
Lynda McGinley, *University of Colorado*

PROCEEDINGS PRODUCTION
Rob Kolstad, *Delos*

EXTERNAL ANONYMOUS REVIEWERS

Lee Amatangelo
Eric Andersen
Paul Anderson
Martin Andrews
William Annis
Steve Armijo
Vidar Bakke
Bob Beck
R. Bernstein
Jon Petter Bjerke
Ian Bland
Mark Burgess
Strata Rose Chalup
James R. Clifford
Alva Couch
Crispin Cowan
Matt Curtin
Kjetil Danielsen
Sven Dietrich
Roger Droz
Chris Faehl
Esther Filderman
Alex Flynn
Doug Freyburger
Luc Girardin
David Greig
David Harnick-Shapiro
Hårek Haugerud
Emmet Hogan
Steve Holstead
Doug Hughes
Tore M. Jonassen
Brian Kirouac
Philip Kizer
Karl Larson
Bert Laverman
Bruce R. Littlefield
Lindsay Marshall
Nadine Miller
Ellen L Mitchell
Adam Moskowitz
Tejas Naik
Evi Nemeth
Alexei Novikov
Kathy Penn
Tom Perrine
S. Rajagopalan
David Ressman
Craig Robersten
Frode Eika Sandnes
Phil Scarr
Michael W. Shaffer
Juergen Schoenwaelder
Willem A. Schreuder
John Sellens
Kent Skaar
Demosthenes Skipitaris
Jon Stearly
Erik Stenvall
Scott Stonefield
Sigmunds Straumsnes
Todd K. Watson
Liza Weissler
Tara Whalen
Ozan Yigit
Elizabeth Zwicky

PREFACE

Welcome to LISA 2001!

Not only is the year 2001 the first year of a new millennium but, thanks to Clarke and Kubrick's legendary Space Odyssey, it has become a symbol of humanity on the threshold of new development. It seems fitting that LISA 2001 should celebrate the occasion, both by looking forward to the future of system administration, as well as reflecting back on the lessons of the past.

LISA remains a conference by system administrators and for system administrators, but it has grown at the periphery and continues to grow, touching on all aspects of computing systems and networks. This year, even as we reflect on past accomplishments, the papers presented here break new ground by applying technologies from diverse and unorthodox areas of science to the problems of network and system management, security and all its related topics.

In registering for LISA this year, you will have received a surprise in your bag, to celebrate the special occasion: a collection of selected papers which summarizes some of the milestones of the USENIX community since the first LISA conference. It is our hope that this volume will not only make interesting reading for future LISA contributors, but will help to cast light on some historical developments for researchers in other fields. It also makes available some older works which have not been in easy reach.

So the tradition now moves on into the next millennium with a new breath of life. Alongside traditional LISA-like discussions of tool building technologies, we have seen a move to more theoretical submissions. I would like to think that I have helped to make this happen, by surreptitiously planting some seeds. These original contributions have their own stories to tell, however. Long overdue, we see ideas from fault tree analysis and scheduling coming into play, alongside modelling techniques and empirical analyses. This is not at the exclusion of more technology oriented topics, such access control management, tools for SNMP queries and relational databases for managing policy agents like cfengine. Some of the topics are archival, possibly dead-ends in the evolutionary tree, while others are just unfolding. The message of 2001 is that we have something to learn from all of these scenarios.

This year the program committee decided to introduce the idea of the "short paper," in order to bring more of the ideas to light under the conference. Papers that we felt could be explained in a short space, with reference to details elsewhere, were placed in a special track designed to celebrate "innovation without elaboration." These papers are of the same high standard as the regular papers and thus can now also find a spot in what was already a packed program.

We received 76 submissions in all. These were funnelled through the most rigorous process of peer review ever, to ensure that every paper was reviewed by several experts, rather than simply random readers. 19 of them survived the ordeal as longer papers, and 11 were accepted as short papers.

There is much to look forward to in the coming week, not only science and engineering. The invited talks track has doubled in size this year; the highly successful Network and Security track now overlaps and augments it with special focus. That's not to mention all of the usual social events. It's a rollercoaster with almost no opportunity for respite! I hope this proceedings will allow you to reflect on the depth of the contributions in the quieter moments after the conference.

As always, the Usenix staff deserves everyone's gratitude for their boundless professionalism and enthusiastic support. LISA is simply the best conference anywhere.

Mark Burgess
Program Chair

Defining the Role of Service Manager: Sanity Through Organizational Evolution

Mark D. Roth – University of Illinois at Urbana-Champaign

ABSTRACT

In large university environments, a centralized academic computing organization is often responsible for providing campus-wide computing services. These organizations usually understand the need for system administrators and software developers, but the people they hire for these roles will often wind up managing service software as a secondary responsibility. This approach often results in poor service quality, overworked employees, and high staff turnover.

In this paper, I present the role of service manager as I have helped define it in my organization and explain how it addresses quality of service issues and staffing shortages. I also detail the organizational changes that my department has implemented to create a group of service managers, relate the process of implementing those changes, and examine some of the pitfalls that were discovered in the process.

Through the changes described in this paper, we have improved our quality of service, dramatically reduced the frequency of late-night pages, quadrupled the number of systems we can run with a fixed-size staff, and greatly increased staff retention.

Introduction

The Computing and Communications Services Office (CCSO) is the centralized academic computing department at the University of Illinois at Urbana-Champaign (UIUC). When I first started at CCSO in 1998, there were two groups that were involved in running most of our production services.

The Unix Systems Team (UST) consisted of 3-4 Unix administrators who were primarily responsible for the Student/Staff Computing Cluster, a group of Unix machines that students and staff could log into remotely to read email, access news groups, store files, etc. Because these services were tied so closely to the operating system, the same group of people maintained both the machines and all of the custom software running on them.

The second group involved in running CCSO's production services was the Software Development Group (SDG). SDG was supposed to be responsible for developing software as needed for CCSO's production services, but the developer of a given service usually ended up running the service once it went into production.

There were a number of problems with this organizational approach:

1. The primary responsibilities of the existing groups were system management and software development, and the costs of these tasks were well understood. However, there was no single group whose primary responsibility was to plan, deploy, and manage production services, so the cost of these tasks were completely hidden. In particular, no one budgeted for the additional staff needed to run each service.
2. Because the existing groups were expected to manage services in addition to their primary responsibilities, the staff was extremely overworked and undervalued. This exacerbated the already difficult problem of hiring and retaining good IT people in academia.
3. Even if the existing staff had not been overworked, managing production services was not part of the core competency of either of the existing groups. For system managers, knowing how to manage Unix systems is different than knowing how to run an LDAP server or a news server. For developers, the need for blocks of uninterrupted time to develop software is inconsistent with the time demands of maintaining a production service.
4. Because most of our services were being run by the developers directly, there was very little separation of production and development. This made it extremely difficult to track changes and keep our services stable.

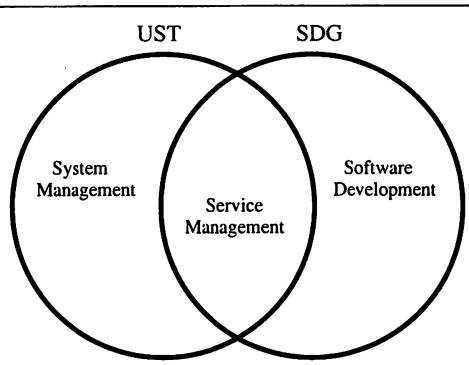


Figure 1: Venn diagram of original group responsibilities.

I first joined CCSO as a member of the Workstation Services Group (WSG), which provides support for Unix workstation owners in other departments on campus. One of WSG's most successful services is the contract administration program, in which Unix workstation owners pay WSG to manage their systems for them on a per-system basis. At the time, WSG was not involved in running CCSO's production systems, but we had built a reputation as a talented and effective group of Unix administrators, and management had taken note.

A New Approach

I first became involved in examining our method of managing production systems in mid-1999. WSG was approached about helping UST with some Student/Staff Cluster upgrades during one of their staffing shortages. That round of upgrades went very well, which led to discussions about how we could be of assistance on a more permanent basis. After reading papers like "Bootstrapping an Infrastructure" [1], I felt that we could improve things by forming a new Unix administration group based on long-term thinking, standardization, and scalability. The result was the formation of the Production Systems Group (PSG) as a subset of WSG.

(Another common way of addressing staffing shortages in academia is to make use of undergraduate labor, as described in "Guerrilla System Administration" [2]. It should be noted that we do employ a small number of undergraduates, but we cannot use them to directly manage our production systems, since they cannot be expected to be on call on a 24x7 basis.)

As PSG was getting off the ground, we quickly realized that we couldn't achieve the scalability we wanted without defining our relationship with our customers. As we started to set the boundaries of this interface, the need for a new service management group became apparent.

The Service Manager

Our customers on campus see the services we offer directly; they don't see system administration or software development as things that affect them. As a result, we needed to define a role whose primary responsibility was not systems or software, but services – in effect, this is the person who's looking out for the end-users' best interests, since they are his customers. Using a term mentioned in "Deconstructing User Requests" [3], we called this person the service manager.

An individual service manager is responsible for planning, implementing, and maintaining a given production service, including the following activities:

1. Initial Investigation
   - Work with management to develop a business plan and list of specific requirements for potential production services.
   - Evaluate software options that meet the requirements, including buying commercial software, using open source software, and developing new software locally.
   - Based on the software evaluation and on input from the system managers and the Operations Center, choose the best alternative for implementation (both hardware and software).
2. Production Deployment
   - Install and configure the service software on the hardware configured by the system managers.
   - Produce internal documentation of the production service configuration, including operational procedures such as backing up and restoring service-related data, adding and deleting users, etc.
   - Coordinate with the Documentation Group to produce end-user documentation for the production service. Also work with User Services to ensure that they're prepared to answer questions from users of the service.
   - Designate and train a backup service manager for the production service.
   - Coordinate monitoring activities for the production service with the Operations Center.
   - Coordinate the exact date and time that the service will go into production with the Operations Center and the system managers.
3. Ongoing Maintenance
   - Coordinate ongoing maintenance of the production service with the system managers and the Operations Center.
   - Coordinate with User Services and the Documentation Group to communicate service changes to the user community.
   - Respond to problem reports from User Services or the Operations Center in a timely fashion.
   - Maintain a list of desired enhancements to the service that can be addressed during the next development cycle.

As a direct consequence of this definition of the service manager's role, the new structure that we were targeting for our entire organization quickly became clear (see Figures 2 and 3).

Advantages

In early 2001, we got approval to form the new Production Applications Group (PAG) to take on the role of service manager. The group initially consisted of three positions, one of which was the manager of the group. Although the group is still in its infancy, we are already seeing several key benefits.

Economy of Scale in System Management

Because PAG is taking on the role of service manager, PSG has been able to achieve the scalability

we were striving for. We've been focusing on building a scalable systems infrastructure, standardizing our procedures and system configuration process, and developing tools to automate as much of our work as possible. Because we spend time anticipating and avoiding problems instead of fighting fires, we've been able to provide a consistent, up-to-date environment and stay responsive to customer requests.

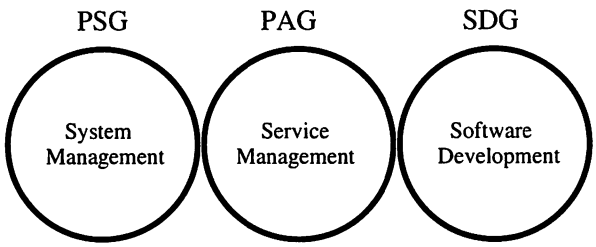


Figure 2: Venn Diagram of current group responsibilities.

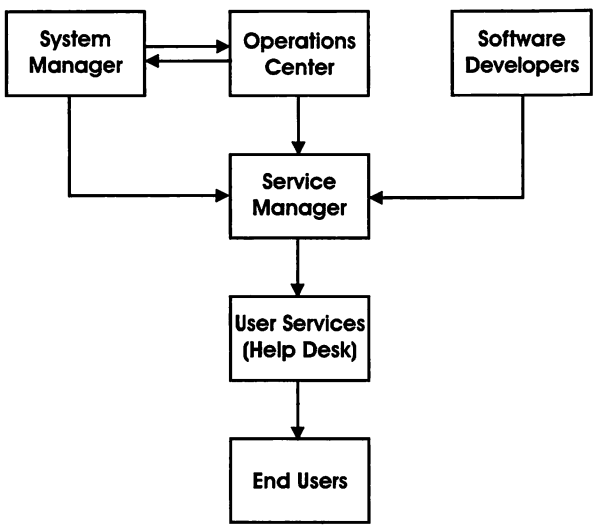


Figure 3: Desired CCSO organizational structure (arrows point from provider to customer).

Clarification of the Developer's Role

Just as PSG's existence helped define the role of the service manager, PAG's existence has served to clarify the role of the software developers in SDG. The software developer's role is to work with the service manager (who is his customer) to do any in-house software development necessary for the creation and integration of a service. The service manager provides a list of written requirements, and the developer is responsible for designing, implementing, and documenting the service software before handing it off to the service manager.

Because they are now able to focus on development, SDG is in the process of standardizing their development practices. They are beginning to think about revision control systems, code reviews, documentation, best common practices, and other development infrastructure standards.

Easing of Staffing Problems

Basic psychology proves that people are happiest when they have authority and responsibility in equal measure. The division of labor between system managers, service managers, and software developers has allowed us to define the scope of each group so that they each have authority and responsibility over their respective arenas and well-defined customers who provide feedback. These changes have increased job satisfaction across all three groups, which has helped tremendously with staff retention. Making our positions more desirable has also made it easier to hire new staff.

Improved Communication and Customer Focus

The process of creating PAG and clarifying the roles of PSG and SDG has allowed each group to focus on who their customer is. This improves our responsiveness to end-users, provides a higher quality of service, and helps ensure that everyone's needs are being provided for. The result is a higher overall quality of service: Services aren't disrupted very often, and they're fixed quickly when they do break.

Providing higher-quality service has also led to more positive feedback from both customers and management, which has been another boon to job satisfaction.

Implementation

The process of creating PSG and PAG has moved very quickly by organizational standards. It started in the summer of 1999 with the formation of PSG as a subset of WSG. For the next year and a half, PSG's focus was on developing our infrastructure and gaining acceptance. In the spring of 2001, as PAG was formed and began to develop its own infrastructure, PSG became a distinct group in its own right.

Getting Buy-In

Because the formation of PSG was worked out with the cooperation of management, we didn't have to worry about getting additional buy-in from that level. However, it was extremely important to get buy-in from the developers (who were managing our production services at the time), since they were to be the bulk of our customers. Even if we had been in an environment where management could force the developers to work with us, we needed to establish a cooperative relationship with them to keep everything working smoothly.

Our general approach to getting buy-in from the developers was to prove to them that we were able to be more responsive than UST had been and that we were able to make their lives easier. Each time we dealt with a new developer, we sat down with them and explained what we were trying to accomplish and listened to their requirements. If they were apprehensive about any of the changes that we were planning to make, we proposed that they work with us to build the

new environment, and that if they weren't pleased with the result, we wouldn't go production with it. (We have not yet encountered a case where the customer wound up taking us up on this.)

Build on Existing Strengths

WSG had several important strengths that we were able to build on for PSG.

WSG's greatest strength has always been its hiring and training practices. Instead of looking for candidates with extensive experience, they look for candidates who may have less experience but have good communications skills and the ability to learn. When a candidate without very much experience is hired, they work closely with existing employees to learn what they need to know.

This approach is important for several reasons. First, Unix administration can be taught, but communications skills and the ability to learn cannot be. Second, when existing employees work with new employees, it helps create a cooperative group culture. And finally, it alleviates the classic problem of hiring seasoned system administrators in an academic environment or in a tight job market.

Because PSG was initially a subset of WSG, we inherited WSG's hiring practices, and they became our strongest advantage. Much of what we have accomplished since would not have been possible if we had not been able to hire the right people.

Another important strength of WSG is that as a cost-recovery group, they depend on paying customers for continued funding, so they have developed a culture of good customer service. They also have the freedom to evaluate new work as it comes in and decline to take on new work if they feel that they can not do a good job in a particular environment. Taking this same approach in PSG allowed us to implement the group in an incremental fashion.

Incremental Approach

The most important part of this transition is that we've implemented it in an incremental fashion. For example, instead of PSG assuming responsibility for a large number of existing UST machines all at once, we waited until each machine was due to be upgraded. When that occurred, PSG worked with the existing service manager to build a new environment on a system with a standardized PSG configuration. Once the new environment was ready, it replaced the old environment in production, and PSG became responsible for it.

This approach has been important for several reasons:

- The existing UST systems had no standardization and no documentation, so PSG would have needed to reinstall them anyway. Waiting for an upgrade allowed us to avoid unnecessary service disruption.
- Because it took time for us to build our infrastructure, PSG had relatively limited capacity when the group was first formed. As we built our infrastructure, we had the capability to take on more systems. The incremental approach allowed us to take on systems as we felt we were ready for them.
- Each success that we had was visible to those customers that we were not yet working with. The enthusiasm of our existing customers made them very useful as references when we needed buy-in from later customers.

One Problem at a Time

One interesting point about the process of forming PSG and PAG is that we didn't originally intend to address the service management issue. We were only thinking about addressing our system management problems when we formed PSG. It wasn't until after those problems were addressed that the need for service managers became clear. In effect, our approach was to solve one problem at a time, since each solution helped us determine where to shift our attention next.

Forming the groups one at a time had several major advantages. First, we were able to focus all of our attention on each group as it was first being formed and developing its infrastructure. Second, we avoided a lot of organizational confusion which could have been caused by making two sweeping changes at once. Third, because PSG's need to define its relationship with its customers was one of the main factors that led to the formation of PAG, people were already familiar with the service manager's role when PAG was formed, so getting buy-in for the new group was much less of a problem than it had been for PSG. And finally, our success with PSG made it easier for management to obtain the necessary funding to create PAG.

Focus on Hiring Good People

Just as it was crucial to hire the right people to populate PSG, it has also been extremely important to have the right people in PAG. In particular, the manager of PAG was chosen from an internal search, since we felt that it was important for this role to be filled by someone who was already familiar with our organization.

In addition to the manager, PAG has hired three additional Service Managers from a mix of internal and external searches. They are individuals who have both technical expertise and project management skills. Because PAG is still in its infancy, it is too early to tell how heterogeneous the group will become. However, it is expected that the group will eventually be quite large due to the number of services that it will be responsible for, especially considering that multiple people within the group must be trained to back each other up on each service.

Pitfalls

There were several pitfalls in the process of implementing PSG and PAG. We anticipated and

avoided some of them, but others needed to be handled on the fly. Thus far, none of these has been a show-stopper.

Root Access

The aspect of PSG's system configuration that caused the most concern from the developers was that we did not provide root access to service managers on our production systems. Instead, we worked with the customer to define the privileged tasks that they needed to perform on a routine basis and allowed those tasks to be accomplished via a mechanism like sudo [4]. Most of the developers were used to having root access on the production systems, and were reluctant to give up that access. These fears were usually based on previous experience with system managers who were not able to be responsive to customer requests. However, once we demonstrated to our customers that we did respond quickly to requests, they were all quite satisfied with the new arrangements.

The System Manager/Service Manager Interface

One of the hardest parts of defining the service manager's role has been explaining to people where the system manager's responsibility ends and the service manager's begins. In some cases, such as an LDAP server or a news server, the distinction is easy to identify; the LDAP or news server software and data are the responsibility of the service manager, and everything else is the responsibility of the system manager.

However, in other cases, the distinction is much less obvious. For example, in the case of a mail server, should the service manager be responsible for sendmail even though sendmail has traditionally been part of the system software? The answer to that question really depends on the details of the service, so we handle it on a case-by-case basis. Before taking on a new system, we will sit down with the service manager and come to an agreement about where the line will be drawn for that particular system and service.

There are no hard and fast rules for where we draw the line, but the most useful guideline is economy of scale. Whenever possible, a facility that is common to more than just one or two services should be handled by PSG to capitalize on our economy of scale. For example, PSG has a standard Apache installation that we can install upon customer request, since a large number of the services that run on our systems are web-based. On the other hand, we have one customer who does a lot of coding in Java, but we don't offer any support for this (other than simply installing the Java packages from the OS vendor) because none of our other customers require it.

The Service Manager/Developer Interface

The classic problem with separating development and production is that the developers don't have direct understanding of what makes an application easy to maintain in production. We hope to avoid this problem by carefully defining the interface between the Service Manager and the developers.

When a service manager determines that there are no acceptable existing software packages, he or she may contact a service developer to request that in-house development be done. The developer's customer is the service manager, who must give him a written list of requirements for the service. Based on those requirements, the developer is responsible for designing, implementing, and documenting the service software before handing it off to the service manager.

It should be noted that once the developer's completed software is handed off to the service manager, it should be treated no differently than if an off-the-shelf commercial product had been chosen. The documentation written by the developer should describe the installation and configuration of the software in a generic form, not the local settings and operational procedures actually used by the service manager to manage the service. The latter will still need to be written by the service manager prior to deployment.

Providing Job Satisfaction

It's been suggested that good technical people prefer a mix of fire-fighting and project work. While our environment is moving in the direction of avoiding the need for fire-fighting, we recognize that we can't anticipate every possible problem, so there will always be a small fire-fighting component of both our PSG and PAG positions. Also, because fire-fighting is only a small component of our environment, we have focused our hiring practices to find candidates who enjoy project work more than they do fire-fighting.

Another concern was that PSG's standardized system configuration would make the system manager's job less interesting. However, our ongoing work on new automation tools and standardized system configuration provides enough interesting job content to compensate for the lack of flexibility on individual systems, so this has not been a problem.

Thoughts About Other Environments

The process of implementing PSG and PAG was tailored for an academic environment like ours. In this section, I will theorize as to what might be different in an enterprise or ISP setting.

Getting Buy-In

In a commercial setting, management typically has more authority to issue directives to those working in the trenches. As a result, getting buy-in from management becomes a crucial part of the process. However, getting buy-in from the technical people is still crucial, since organizational changes will not work very well if people are forced to implement them against their will.

Implementing the Details

While a directive to create a service manager role may come down from management in a commercial

setting, the details of how to implement the change should still be decided at a very low level. This would help people feel as though they have a hand in the decision-making process, which helps with getting buy-in.

Which Comes First?

In our environment, it was the formation of PSG that put the spotlight on the need for PAG. However, in an environment which already has a good group of system managers, it might make more sense to leave them as-is and focus immediately on creating a group of service managers. This should be fairly easy to sell to the existing system managers by focusing on the fact that the service managers will help ease their workload.

Change Control

One of the tasks that both PSG and PAG are working on is to implement change control procedures for our production systems and services. Our goal is to focus on the beneficial aspects of change control without becoming too mired in bureaucracy.

Many corporate environments already have complex, bureaucratic change control procedures in place. The creation of a group like PAG may allow the change control procedures to be simplified to maximize benefit and minimize overhead.

Conclusion

The organizational changes we have implemented to create a service management group have had excellent results. In the last couple of years, we have experienced a huge growth in the number of production systems we run. UST used to manage approximately 20 systems with five people; PSG is now managing approximately 85 systems with 4.5 people. The average turnover time for one of UST's high-profile positions was approximately one year; in contrast, PSG has existed for over two years and has never lost an employee. Emergency pages have gone from almost nightly to almost never. Finally and most importantly, the quality and reliability of our production services has improved noticeably.

Acknowledgements

I would like to thank Mona Heath not only for her proofreading, grammatical corrections, and suggestions on content, but also for her overall mentorship over the last several years. Without her support and encouragement, none of the events described in this paper would have been possible.

Author Information

Mark Roth is the manager of the Production Systems Group of the Computing and Communications Services Office at the University of Illinois at Urbana-Champaign, where he earned a Bachelor's degree in Computer Science. Mark is also the author of several open-source software packages. He can be contacted via email at roth@uiuc.edu, and his web page is http://www.uiuc.edu/ph/www/roth.

References

[1] Traugott, Steve and Joel Huddleston, "Bootstrapping an Infrastructure," *LISA XII Proceedings*, Boston, MA, pp. 181-196, 1998.

[2] Hunter, Tim and Scott Watanabe, "Guerrilla System Administration," *LISA VII Proceedings*, Monterey, CA, pp. 99-105, 1993.

[3] Limoncelli, Thomas, "Deconstructing User Requests and the Nine Step Model," *LISA XIII Proceedings*, Seattle, WA, pp. 35-44, 1999.

[4] Miller, Todd, "sudo," http://www.courtesan.com/sudo/.

Remote Outsourcing Services for Multiple Branch Offices and Small Businesses via the Internet

Dejan Diklic, Venkatesh Velayutham, Steve Welch, Roger Williams
– IBM Almaden Research Center

ABSTRACT

Maintaining a reliable computing infrastructure encompassing servers, clients, printers and Internet connections is one of the major problems faced by branch office managers and small business owners. The cost associated with good systems administration is too high for either small businesses or branch offices. At the same time large service corporations don't see this market as attractive due to the high startup and infrastructure cost. We at IBM Almaden Research Center developed a research prototype that enables remote outsourcing services for multiple small offices from a central location via the Internet using the already available commercial software. This technology provides a solution for both small business owners (branch office managers) and service providers. We use the term small business for any company with one to 100 client machines. For our purposes branch offices can be categorized as small businesses. By combining our newly developed systems management technologies with an existing Internet connectivity server, small office LANs can be remotely monitored and managed via the Internet. At the same time the need for expensive leased lines and dedicated routers for secure connectivity is removed since both routing and security are provided through software. In this paper we will describe the approach taken as well as utilized software and hardware components.

Now that we described the target environment we introduce all of the components of the solution. In the second section we describe implementation details. Short summary of the work is given at the end of the article.

Introduction

Most of readily available network management applications such as Tivoli ITDirector and HP Open View [1, 2] provide all usual computer management functions such as: remote control, software/hardware inventory, task scheduling, sending alerts, logging events and software distribution. When combined together these functions represent a set of tools that is used by various corporations to manage their internal network with thousands and tens of thousands of clients [4, 7]. These management applications are installed on a server which is on the same network as the client machines they manage [5]. To put it simply, "if you can ping the client you can manage it." While this works for internal networks of large corporations it is not usable if the client machine is connected on a different subnet and/or hidden behind the firewall inside a small business or branch office. The main goal of our research project was developing software extensions to regular off-the-shelf network management applications which would enable management of client machines over untrusted networks such as the internet.

The approach taken with the purpose of addressing these issues started from a very simple idea. If we can use one server located at the central network operations center (NOC) of the service provider to manage multiple small businesses (without a server at the customer site) the cost for managing each business will be much lower then when using a regular management model described above. Another issues addressed by this model are the ease-of-use for the customer and the ability to outsource system management. By utilizing this model management of IP-disjoint pools becomes possible. This approach applies to both small businesses and large corporations that use firewalls on the Intranet to separate networks. Another advantage of the new approach is that a service organization will not have to own or maintain the server and server software at customer premises but only at the central location. Most small businesses can not afford the cost of a management server, but they can afford the clients and associated software.

In Figure 1 we present the view of the network architecture as is typically seen by the network management providers [6, 8]. Management server which is located on the side of corporate Intranet is usually a very powerful machine. On the other side of the network (Internet) are multiple small businesses and branch offices that need management services. Small businesses or branch offices are typically setup to run their own networks using non registered private IP addresses. They are connected to the Internet through an Internet gateway which can be anything from a

regular PC to a small business server such as IBM Whistle InterJet II, Cobalt Qube, etc. We assume that both the service provider and the small businesses are isolated from the Internet by a firewall.

This architecture makes it almost impossible to remotely outsource to small businesses and branch offices [11]. Since most of the SMBs are using the same private IP address range, the management application has to not only traverse two firewalls, but has to access machines hidden on the private network. As part of this project, we set out to develop a new network protocol with an advanced addressing schema which would allow us to traverse multiple networks, firewalls and duplicate IP addresses while at the same time keeping the system secure.

We attacked several issues in order to solve the problems described above. The first issue was that of addressing different machines with the same IP address on different private networks. To solve it we developed a novel form of network address which enables us to both access the machine at the private network behind the internet gateway and at the same time keep the network infrastructure secure from outside tampering. In addition to that packets are encrypted across the untrusted environment and delivered through a firewall to the addressed machine without compromising the small businesses network security. The next challenge was related to changing as little as possible of the original management software. Due to the architecture of the software used (ITDirector) this was rather easy to achieve. Another requirement was that new system has to use existing holes in the firewall and existing hardware at remote sites (no new hardware required).

Solution

In the next three chapters we describe the main idea behind the solution in more detail. First we describe the extended network addresses needed to address all of the machines in disjoint IP pools. Then we describe newly developed (java-based) proxy which runs on the Internet gateway machines. The last chapter is dedicated to security problems inherent in a system like this.

Network Address

The first part of the solution involved changing the network management protocol so that more complex network addressing schemes could be used. To maintain the simplicity of the SMBs LAN, private dynamic addressing schemes were widely used. Therefore, regular four byte IP address can not be used to identify a particular machines in the private addressing space. We devised the way to address the target machine by incorporating the IP address of the Internet gateway.

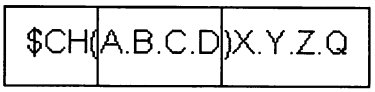


Figure 2: New network address format.

While the four byte IP address will suffice once the packet is delivered to the internet gateway, we need a way to address at the same time both the internet gateway and the client machine. The new format of a new network address is shown in Figure 2.

The regular IP address of a machine on the private network (X.Y.Z.Q) is combined with the external

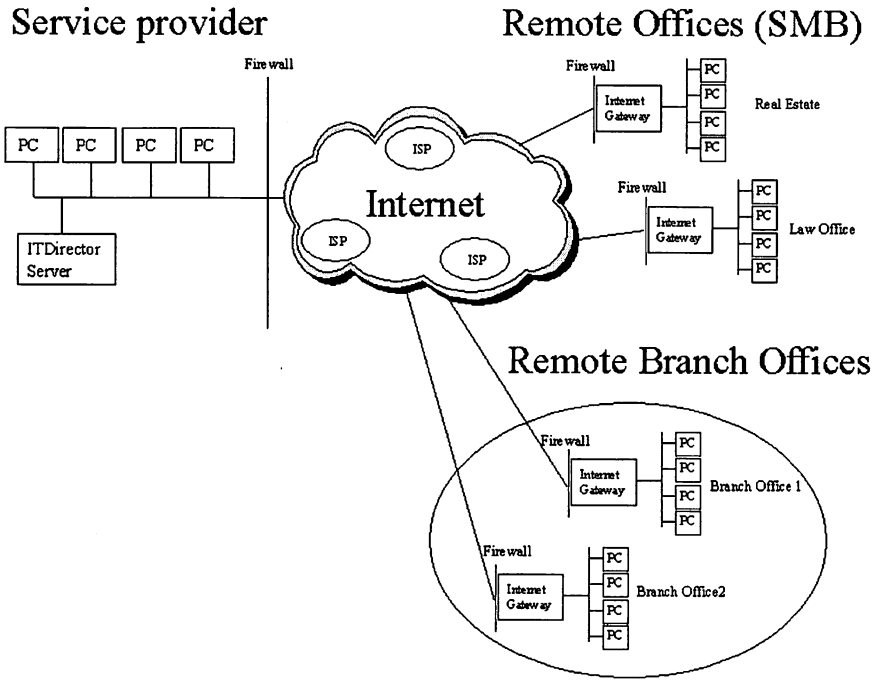


Figure 1: Typical network architecture.

registered address of the Internet gateway (A.B.C.D). This way, the packets are first delivered to the gateway with address A.B.C.D. Then they are forwarded to the private network to the machine with the address X.Y.Z.Q. While the address of the from shown in Figure 2 is user readable, it is not used in such a form in the network protocol to reduce overhead. We encode the addresses as an eight byte array at the end of the network packet. This minimized the overhead associated with the sending of every packet when compared to using the address shown above. At the same time the processing time required for packet forwarding is minimized. When this new complex address form is introduced, it becomes possible to address machines that are more then one network hop away from the server. Any extensions of this type of network addresses are easily done. If we need to deliver a packet to a machine which is three hops away from the server, extending the protocol would be just a matter of attaching four more bytes to the end of the package.

Gateway Proxy

Having solved the problem of addressing client machines behind the Internet gateway machine, the problem of delivering packets to a client machine behind the firewall needs to be attacked. There are several options available. The fist one that comes to mind if also the simplest one. It requires a hole to be opened in the firewall of the Internet gateway machine. Then a proxy application can be installed on the gateway machine that will accept packets coming from the newly opened hole in the firewall. Both the packets originating at the server and at the client will be directed to the same port and then consequently forwarded to their destination by the proxy application running on the gateway. Most of the small businesses are however not willing to expose themselves any more then necessary and are highly critical of opening holes in the firewall. Therefore we developed an alternative method to traverse firewalls. We made several assumptions about small businesses. We assume that they are connected to the Internet and that the gateway machines come installed with a web server capable of running CGI scripts [7, 12]. In order to pass the packet through the firewall we decided to pass the packets through the web server using our novel CGI script. While this way of distributing packets is rather slow, when compared to the dedicated application sitting on a dedicated open port, it enables us to solve this problem without needing customers permission to open an additional hole in the firewall. Any communication between the server and CGI application and client and CGI application is performed using standard GET and POST methods. To pass network packets we developed an application that registers itself to a predefined URL such as http://A.B.C.D/my_cgi_app where A.B.C.D is the address of the Internet gateway device. The CGI application will receive packets coming from the Internet (server) and then based on the address of the private machine which is read from the network packet itself, forward the packet accordingly. The end-clients address is stripped off from the packet before it is forwarded so the client can process it. This proxy has several different functions. The most obvious one, which is forwarding packets from the server to the client and vice versa was described above. The second function is described below.

The Internet gateway machine which is running the proxy application has to be detected from the management server for the system to be fully usable. The detection mechanism built in network management applications can be rather complex. We tried several different approaches to implementing the detection mechanism in the proxy application. The first approach was to try using hard coded responses to discovery packets. This was too complicated and resulted in unreliably functioning proxy. The second approach yielded much better results. Since the code for the java based agents is available, it is possible to add the forwarding and complex network address thereby effectively creating a detectable proxy that forwards packets and understands the long network address.

Security

One of the problems faced by service providers is ensuring security and confidentiality of data while remotely outsourcing the customers devices across an untrusted network (Internet). We added security to remote outsourcing by establishing an encrypted session (Virtual Private Network) [5, 9, 10] between the service provider and the internet gateway. We use SSL and client side certificates to make sure that nobody else but authorized users can connect to the gateway proxy [3, 14]. Since the Internet is not a trusted network, the remote session management data and protocol is sent through the VPN [13, 15]. After the packet is delivered to the proxy machine it is decrypted and forwarded across the local area network to the end-client(s). Since the local area network of the small business or the branch office is trusted, the data is sent unencrypted. Our design also facilitates the packet sent encrypted all the way to the end client. This addition to the system is essential since remote outsourcing could result in having confidential data exposed to the outside world and prove damaging to most businesses.

The biggest problem when dealing with encryption is the need for powerful processors that can complete encryption and decryption of large amounts of data in a very short time. If the processor is slow the encryption/decryption will take a long time and the whole remote management process is endangered since the operator can't perform tasks in the real time. A delay of several seconds is not acceptable when performing base operations (like clicking the mouse button, or opening the window). There are several ways this problem can be solved. The easiest is to make sure that the processing capability of the Internet gateway is sufficient. This is highly expensive and it increases

overhead added to each device. Another way is to add encryption acceleration hardware to the device which is a lot cheaper and equally effective. Additional area that could be easily addressed is the option of specifying tasks that should be performed using encryption and tasks that should be performed without encryption. For example if you are making a backup of financial data, you would want to be as secure as possible, but if you are just distributing the latest version of some software no encryption is necessary. The same holds true for regular everyday monitoring of system events. Encrypting data such as "Disk out of space" messages is in most cases not essential.

All the regular safety measures provided by ITDirector are still present in the new protocol. Users need to be registered to access all of the functions of the server and clients can be controlled only if the proper authentication is provided to the client (once the client is locked).

Implementation

Since there are several systems management applications on the market at the beginning of the project we debated pros and cons of IBM Tivoli ITDirector and HP Open View [1, 2], as basis for our prototype. We selected ITDirector since it has a rich set of utilities and functions to manage all aspects of the PC environment and is used and developed by IBM. ITDirector enables us to utilize the rich set of APIs and extend network protocol that allows us to connect to clients over the internet using the provided tool kit. ITDirector is shipped with various agents for various operating systems such as Windows NT/2000, 9X, Linux, AIX, OS/2 etc.

The agents for Linux and AIX are available in Java which makes it simple to use as basis for our proxy code. Since the proxy will run on a Internet gateway machine we prototyped our system on several different hardware platforms. Our first prototype was developed on a regular PC that is running Linux and is used as Internet gateway. This significantly simplified the development when compared to developing on a dedicated Internet gateway.

Once the first version was available we integrated and tested it with IBM's Whistle InterJet II and Cobalt Qube2. Whistle InterJet II is a thin server offered by IBM as an Internet gateway. InterJet II is a small footprint, easy to use and simple to administer server based on FreeBSD operating system which connects small businesses to the Internet and provides all of the functions of a file, web, mail, DHCP and DNS servers. Cobalt Qube2 provides the same functionality as IBM's device. To utilize InterJet II or Cobalt Qube2 we had to develop all the pieces needed to incorporate them into the ITDirector addressing and management schema.

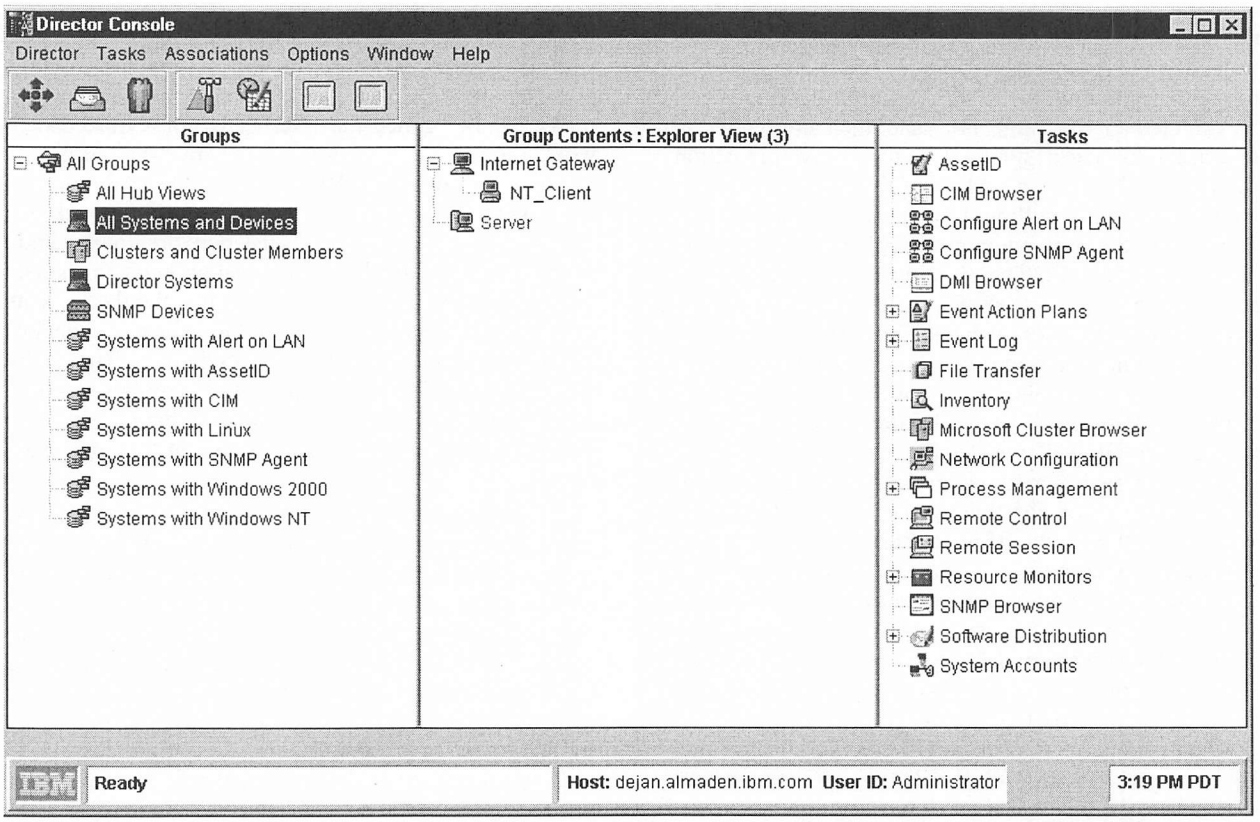


Figure 3: Tivoli TIDirector showing server side extensions.

The addition of the complex network addressing schema and security layer to the network protocol changes the complete network protocol layer of the regular Tivoli ITDirector. This rather complex and significant changes can be done due to the architecture of the product, which makes it possible to just change tcpip.net file which defines the complete network protocol. Security layer was added to the protocol through usage of OpenSSL. The java based Linux agent was the origin of the CGI proxy. Adding forwarding to it made it usable on the Internet. We added security to the java agents by using jsse-ssl. Once the whole system was tested to perform the desired tasks, several interesting observations and measurements were made.

During the performance testing phase we noticed that simply adding one more hop in the network protocol decreased the performance by less then 5%. Adding firewall traversal to the system increased the performance penalty to 8%. The biggest drop in performance however comes when using encryption over SSL. The drop is performance is almost 40%. This means that encryption should be aided by hardware acceleration on the Internet gateway. In Figure 3 is a screen capture of the Tivoli ITDirector screen which shows our extensions to the server side. In the middle column we see only three machines, the Tivoli server, the Whistle gateway, and the client. The presentation of machines on the screen was modified to include hierarchical drawings of associated gateway and clients. In Figure 4. we see the new address of the client machine in the complex form. In Figure 5 we show the grouping of all hubs (Internet gateway machines). This view is very useful since it displays only proxy gateway machines and the clients behind them.

Future Work

At the moment there are several interesting developments going on with this project. The most important one that we are pursuing is software distribution to multiple machines from a central fan-out site (Internet gateway). The idea behind this is that you really need to only once distribute the software packet to the Internet gateway and it can then redistribute the packet to as many clients on the local area network as necessary. This speeds up the process considerably since the big packet is transferred only once over the slow Internet (application specific compression of data). At the same time we are working on speeding up the packet transfer and minimizing the forwarding (processing) time of every packet on the proxy gateway.

Summary

In this paper we presented an extension to Tivoli ITDirector that enables service providers to use ITDirector as the application of choice for managing multiple small and medium businesses from a centralized location. We presented the technical enhancements made to the ITDirector network protocol and the newly developed java proxy for Whistle InterJet II. By

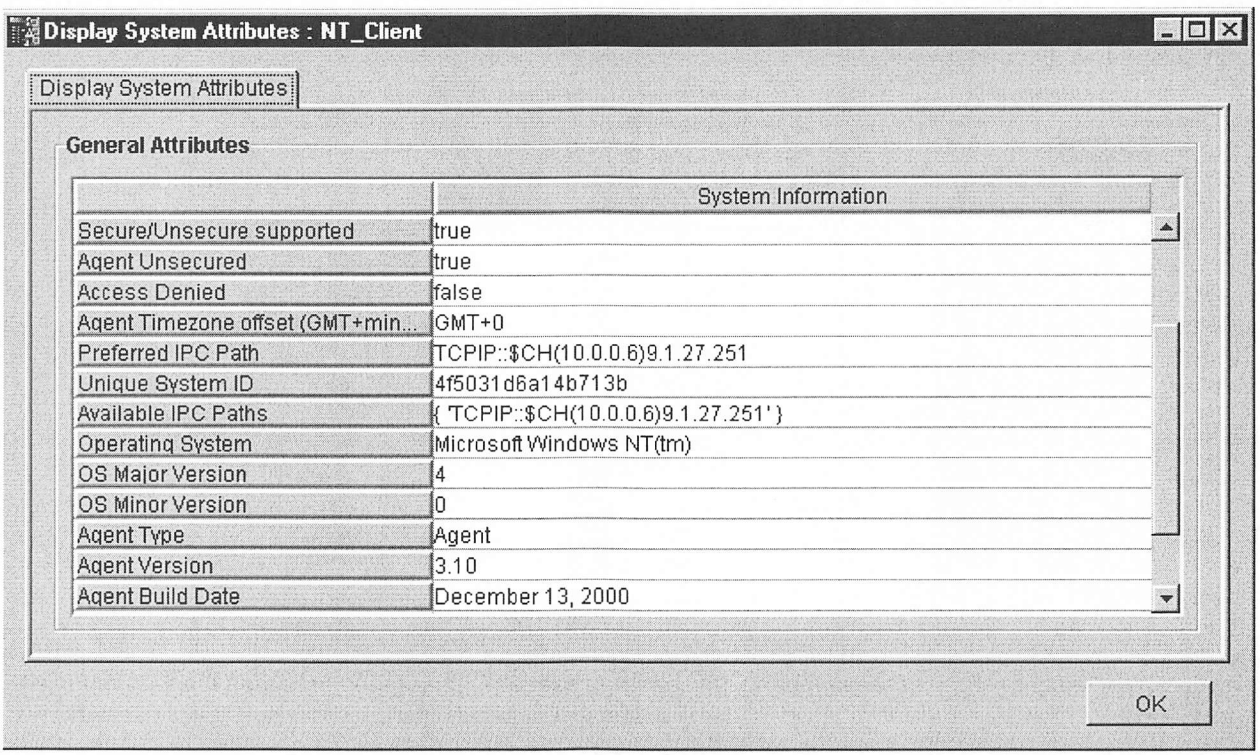


Figure 4: New address of client machine in complex form.

modifying the network protocol by adding one more step in network packet distribution as well as security in form of client side certificates and encryption we proved that the technology for managing multiple small and medium business from a central location can be achieved.

Author Information

Dejan Diklic has a MS in physics from University of Bremen, Germany and MS in computer science from Santa Clara University, USA. He is currently employed at IBM Almaden Research Center where he works on various problems related to systems management and storage. He can be reached at ddiklic@almaden.ibm.com .

Benjamin Reed has a PhD in Computer Science from the University of California, Santa Cruz. He works at IBM Almaden Research Center in the area of network attached storage, pervasive computer systems, and system management. He can be reached at breed@almaden.ibm.com .

Venkatesh Velayutham has a BS in Mechanical Engineering from Anna University, Chennai, India. He is currently employed at IBM Almaden Research Center where he works on various problems related to network management and storage. He can be reached at vvelayutham@almaden.ibm.com .

Steve Welch completed BS degree in CS from Cal-Poly San Luis Obispo. He is a manager of Cyberspace Technologies at IBM Almaden Research Center focusing on development of system management and Internet technologies. He can be reached at swelch@almaden.ibm.com .

References

[1] Tivoli Web site, http://www.tivoli.com/products/index/it-director/ .

[2] HP Open View web site, http://managementsoftware.hp.com .

[3] D.Zeltserman, G.Puoplo, *Building Network Management Tools with Tcl/Tk,* Prentice Hall, 1999.

[4] Mark Burgess, *Principles of Network and System Administration*, John Wiley & Sons, Chichester, 2000.

[5] S. Brown, *Implementing Virtual Private Networks*, McGraw Hill, 1999.

[6] M. Subramanian, *Network Management: Principles and Practice*, Addison Wesley, 2000.

[7] T. C. Mann-Rubinson, K. Terplan, *Network Design: Management and Technical Perspectives*, CRC Press, 1998.

[8] K. Terplan, S. Zamir, *Web-Based Systems and Network Management*, CRC Press, 1999.

[9] Peter D. Rhodes, *Building a Network: How to Specify, Design, Procure, and Install a Corporate LAN*, McGraw-Hill.

[10] H. Hegering, S. Abeck, B. Neumair, *Integrated Management of Networked Systems: Concepts,*

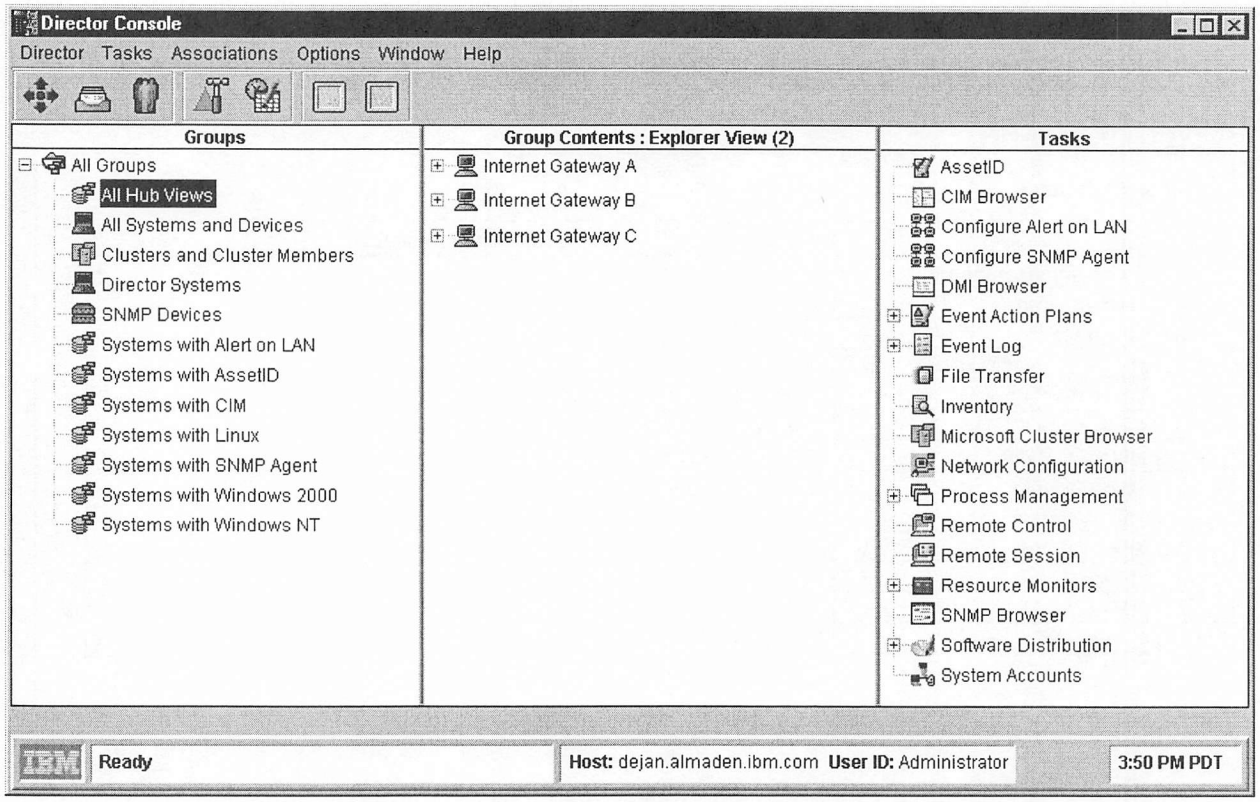


Figure 5: Grouping of all hubs.

Architectures, and Their Operational Application.

[11] M. Subramanian, *Network Management: Principles and Practice*, Addison Wesley, 2000.

[12] K. Terplan, S. Zamir, *Web-Based Systems and Network Management*, CRC Press, 1999.

[13] *The Basics Book of OSI and Network Management by Motorola Codex*, Addison Wesley, 1999.

[14] G. Held, *Managing TCP/IP Networks: Techniques, Tools and Security*, John Willey & Sons, 2000.

[15] A. Leinwand, K. Fang-Conroy, T. Stone, *Network Management*, Addison Wesley, 1996.

SUS – An Object Reference Model for Distributing UNIX Super User Privileges

Peter D. Gray – University of Wollongong

ABSTRACT

This paper describes a system administration tool which allows a user to run a command as root or as some other user after authenticating. Unlike most other commands of that ilk, SUS attempts to treat the command and its arguments as references to system objects, and allows for relatively powerful matching on the attributes of those objects to determine if the user should or should not be allowed to execute the desired command. In addition, SUS has a mode to help limit the number of setuid utilities needed to provide user services via the web.

General Issues

It has long been known that the "all or nothing" traditional UNIX security model can be a serious inconvenience for many sites [1, 2]. The super user account (usually "root") has practically unlimited administrative powers while normal users are subject to all security restrictions. There are very few system administration tasks which can be performed without super user privileges.

At larger sites, it is common to have a team of system administrators of varying seniority, skill and experience, some performing fairly mundane tasks such as changing forgotten passwords.

Having a large number of people knowing the "root" password is not only inconvenient but is an obvious security hazard. In addition, permitting inexperienced or untrained staff to have access to such high levels of power and functionality poses a real risk to the integrity of computer systems. Simple typing mistakes can cause havoc.

There have been several packages written, both open source and commercial, to address this problem with various degrees of success [3, 4, 5]. To use these systems a user invokes the application (which runs with super user privileges via setuid) and passes arguments to indicate which command they wish to run and what arguments they wish to pass. The application consults a configuration file. Depending on the invoking user and the requested (target) command (and possibly other factors) the application will either allow the request and run the target command or inform the user that they do not have sufficient privileges to run the command they requested.

The criteria used in the above determination can be simple or complex depending on the tool being used. The paper by Hill [3] on the tool PRIV gives an excellent overview of the above issues and lists the capabilities of several such systems, both commercial and open source.

With the exception of PRIV, most such systems fail to examine the arguments of the target command with any degree of sophistication. PRIV offers a rich set of functionality along with some attempt to treat the arguments as references to system objects. It allows for command arguments to be restricted to objects which satisfy simple criteria.

SUS takes some of the ideas of PRIV and extends the concepts by adding the ability to treat all commands and arguments as references to objects. Such objects include users, files, groups, hosts and processes. Each such object has a set of attributes. For example, a file has an owner, a group and a type, etc. Objects on which the command will operate are examined, their attributes retrieved and compared with a selection criteria found in the configuration file.

Only if the selection criteria match the attributes of the objects is the command allowed. For example, it is possible to restrict a user to deleting regular files owned by any member of the group "admin" or only allow them to send a HUP signal to a process owned by a user whose UID is in the range 100-200.

Operation

SUS is installed as a setuid binary. When invoked to run a command it reads a single configuration file. The configuration file is preprocessed via a CPP-like macro preprocessor [9] with many predefined macros to allow for simplification of the control file syntax. For example, it is possible to restrict access to commands by time of day using SUS, even though there is no such capability in the SUS control language. The combination of conditional output in the macro preprocessor plus judicial use of the predefined macros set to the time of invocation provides the necessary functionality.

The presence of the macro preprocessor also allows for an easy mechanism for controlling the various operational aspects of SUS. By setting values of special macros in the control file, all configurable aspects of operation can be easily controlled. For example, the location of the log file can be set by simply setting the macro "LOG_FILE" to any desired location.

The configuration file syntax is similar to the popular sudo [4] product from the University of Colorado. Lines are composed of a user selector and a command selector. Each line is read and the user selector used to determine if this line applies to the current user. If so, the command selector is compared with the target command. If a match is found, the command is executed.

Objects and Attributes

SUS allows for commands (including arguments) and users to be matched as simple regular expressions. However, the real power of the system is its ability to treat command names and arguments as references to system objects. For example, in the command

```
$ sus rm a.out
```

SUS can treat the name "a.out" as a reference to a file object.

The object types supported by SUS are:

- USER, defined by the information returned by the getpwXXX(3) routines. Attributes which can be used for matching are the username, user id, group id, gecos field, home directory and shell. The home directory may be treated as a reference to a FILE (see below) and the primary and secondary groups may be treated as references to a GROUP object (see below).
- FILE, defined by the information returned by the stat(2) system call. Attributes for matching are the type, user id, group id, device, raw device and the file's real name. The owner may be treated as a reference to a USER object (see above) and the group to a GROUP object (see below). The parent directory may be treated as a reference to a FILE, PFILE or RFILE objects (see below).
- PFILE, identical to file except matching is performed on the referenced file's parent directory.
- RFILE, identical to FILE except matching proceeds recursively up the file system tree until a match is found or the root directory is reached.
- GROUP, defined by the information returned by the getgrXXX(3)routines. Attributes used for matching are the group name and group id.
- HOST, defined by the information returned by the gethostbyXXX(3) and getipnodebyXXX(3) routines.

```
// Make sure the target environment excludes
// dynamic linker/loader variables.
#define ENV_DELETE "LD_.*"
// Override the SHELL environment variable
#define ENV_ADD "SHELL=/bin/false"
// Allow user joe to add a user via a script
joe : /usr/local/bin/add_user.sh
// Set up a class of users whose home directory
// is in /home/sales using a CPP macro
#define sales USER(home=/home/sales/.*)
// Any user in the above group of users may change the
// ownership of any regular file to themselves
// (predefined macro SUS_USER) as long as
// the file was owned by another
// person in the sales group.
sales : chown SUS_USER FILE(type=reg, owner=sales)
// Trudy can send a TERM signal to any process whose
// group is "eng" or (because we use RPROC rather
// than PROC) has a process above it in the process tree
// whose group is "eng"
trudy : kill -TERM RPROC(group=eng)
// On the hosts in 130.130.62 subnet, andy can run anything
// he wants except the shells (ANY_COMMAND is a predefined macro)
#define SHELLS "/bin/sh | /bin/ksh"
#define secure-hosts HOST(ip=130.130.62..*)
andy @ secure-hosts : !SHELLS, ANY_COMMAND
// Bruce can run anything in the gurus directory with
// any arguments he likes
bruce : FILE(name=/share/gurus/bin/.*) ANY_ARGUMENTS
```

Listing 1: System capabilities.

Attributes for matching are the names and IP addresses (both V4 and V6).

- PROC, defined by the information available in /proc/<pid>/psinfo. Matching can be performed on the process id, the parent process id, the group id and effective group id, the session id, the user id and effective user id and the controlling tty. The user id and effective user id may be treated as references to a USER object, the group id and effective group id may be treated as references to GROUP objects.
- PPROC, identical to PROC except matching is performed on the referenced process's parent process.
- RPROC, identical to PROC except matching proceeds recursively up the process tree until a match is found or the root of the tree is located.
- REGEXP, extended regular expression matching. This is the default if no object class is specified but may be called explicitly if desired.

All of the above classes except REGEXP have an additional attribute "exists" which matches if the referenced object actually exists. This allows restricting an operation to a existing or non-existing object. For example, you may allow someone to create a new file, but not edit existing files.

Example Configuration

The configuration file syntax is basically defined as:

```
user-selector : allowed-commands
```

The preprocessor [9] is general purpose and macros may be defined to control its operation. In its default configuration it closely resembles CPP. Other styles it can support in terms of macro definition and comments are TeX, HTML and PROLOG.

The fragments of configuration file in Listing 1 demonstrate some of the capabilities of the system. C++ style comments are used for annotation.

Operational Notes

Some caveats are in order:

- all matching is done as strings.
- all string matching is performed as anchored, extended regular expressions.
- command lists are matched left to right and matching stops as soon as a match is found.
- if a command matches the entry in the file, but the match expression is negated, searching stops and the command is not allowed.

Target Command Environment

SUS allows practically complete control of the environment of the target command, including the ability to selectively remove or pass through environment variables from the environment of the invoking user which match regular expressions defined in the control file as well as adding or replacing environment variables with new values.

A current directory for the target command along with resource limits may also be set if required. Signal handlers and masks are set to default settings before the target command is invoked.

Security

SUS has been written with care to avoid any problems with buffer overflows or other potential security problems. Buffer size checks are performed on any operation where overrun could be possible. It checks that the control file is owned by "root" and is not writable by other users. Space for all data structures is dynamically allocated and there are no built-in limits in data sizes other than those set by configuration.

Logging

SUS logs all invocations for any reason, successful or not, using either syslog or straight to a file or both. All aspects of the logging operation are controlled by the configuration file. Information in the log records includes the invoking user, the current directory and the target command and arguments. Optionally, the resource usage and elapsed clock time for each target command may be logged as well.

Timestamps

Normally, SUS will force each user to authenticate by supplying their normal system password. When invoked successfully, SUS stores a timestamp for each user in a system directory, normally the root directory, and (optionally) the user's home directory. If a user has invoked SUS successfully inside a short period (configurable) then the user does not have to authenticate. Storing the timestamp inside a user's home directory allows for SUS to remember invocation across multiple hosts, as long as the home directory is shared.

Timestamps include the username, the user id, a SHA1 [7] checksum of the user's encrypted password and a SHA1 checksum of the actual timestamp itself. The root directory timestamp also includes a SHA1 checksum of a string based on the user's plain text password. This checksum field is empty in the home directory timestamp but is included in the SHA1 checksum of the home directory timestamp. This means that it is impossible to compute the checksum of the home directory timestamp without access to the root timestamp. Thus any tampering with the home directory timestamp is detectable. All checksums are checked to ensure a timestamp is valid.

The authentication step is skipped if the root timestamp is valid and current or the root timestamp is valid and the home directory timestamp is valid and current.

Any change to a user's password, username or user id cause all timestamps to become invalid.

CGI Support (Promiscuous Mode)

It is becoming increasingly necessary to allow users to perform tasks previously only offered if the user connected directly to a host (either via TTY style or X11 protocols) via a web interface. Some simple examples are the ability for a user to change their password or query their disk quota. To solve these sorts of problems CGI programmers often have to resort to setuid wrappers or scripts with obvious security ramifications.

SUS offers the capability to allow selected users to run commands as another user on the system if they can supply the target user's username and password. Effectively, user 1 can run a command as user 2 if user 1 can supply user 2's username and password.

For example, SUS can be configured to allow the web server user (say user WWW) to run a command as a normal user. The user can supply their own username and password, which is passed into SUS. SUS will authenticate the target user, then run a command (possibly a script) with appropriate arguments to perform the required operation. The command is run as the authenticated user. This mode of operation goes by the rather intimidating name of "promiscuous mode."

As an example, a site could write a script to change a user's password if run as that user. A CGI script could call the password changing script via SUS. If the user supplies their own username and password to the CGI script (and hence SUS) the password changing script is run as the authenticated user.

The overall result is that any number of scripts can be written which can run as any user who can supply their own username and password without installing any additional setuid binaries other than SUS. The overall number of setuid utilities is kept to a minimum. Exactly the same sorts of controls on who can run what exist for "promiscuous mode" as when operating normally.

Conclusion and Future Work

SUS allows for very fine grained control over what users may or may not do as root. Arguments to commands are examined to determine which system objects they refer to and the attributes of those objects may be used as criteria to allow or disallow the command.

SUS is under active development. Possible future work includes adding an ability to log a transcript of a user's TTY session with a target command and the capability to reliably prevent shell escapes from target commands such as editors.

Availability

SUS is freely distributable under the GPL and available from http://pdg.uow.edu.au/sus It is developed and tested under SOLARIS 8 but is known to compile and run under LINUX. It contains no dependencies on non-standard libraries.

References

[1] Ritchie, Dennis, "On the Security of UNIX," *UNIX Programmers Manual Volume 2*, AT&T, NJ, USA.

[2] Jordan, Carole S., "A Guide to Understanding Discretionary Access Control," *Trusted Systems (NCSC-TG-001)* September, 1987.

[3] Hill, Brian C., "Priv: Secure and Flexible Privileged Access Dissemination," *Proceedings of the Tenth Systems Administration Conference (LISA 96)*, USENIX Association, Berkeley, CA, USA.

[4] "Sudo," University of Colorado, ftp://ftp.cs.colorado.edu/pub/sudo.

[5] "PowerBroker," FSA Corporation, http://www.symark.com/.

[5] Ramm, Karl, and Michael Grubb, "Exu – A System for Secure Delegation of Authority on an Insecure Network," *Proceedings of the Ninth Systems Administration Conference (LISA 95)*, USENIX Association, Berkeley, CA, USA.

[6] Chris Thorpe, "SSU: Extending SSH for Secure Root Administration," *Proceedings of the Twelfth Systems Administration Conference (LISA 98)*, Usenix Association, Berkeley, CA, USA.

[7] "Secure Hash Standard," *Federal Information Standards Publication (FIPS) 180-1.*

[8] "A Secure Environment for Untrusted Helper Applications," *Proceedings of the Sixth USENIX Security Symposium*, Usenix Association, Berkeley, CA, USA.

[9] "GPP – Generic Preprocessor," Ecole Polytechnique, http://math.polytechnique.fr/cmat/auroux/prog/gpp.html.

IPSECvalidate – A Tool to Validate IPSEC Configurations

Reiner Sailer, Arup Acharya, Mandis Beigi, Raymond Jennings, and Dinesh Verma
– IBM T. J. Watson Research Center NY

ABSTRACT

This paper describes a tool for validating the proper configuration of the IPSEC protocol suite including IKE. The tool validates that two hosts are able to communicate (normal ping functionality) and that this communication is occurring using the proper authentication/encryption transformations as required by IPSEC. IPSEC configuration is very complex, and administrators are often unable to determine if a machine configuration is offering the desired protection. IPSEC and IKE operate in a manner transparent to IP applications; an administrator is therefore unable to check the proper operation of an IPSEC "security association" using traditional IP tools.

Introduction

Security for IP-based networks has become increasingly important: with many companies relying on Virtual Private Networks (VPNs) for distributed intranet, extranet, and remote access services, security requirements have become essential. The IETF has developed security protocols and mechanisms that extend conventional IP services by security services [5, 10, 3].

The IP security protocols (IPSEC) are used to encapsulate IP data packets (tunnel mode) or their payload (transport mode). Two protocols are standardized: the IP Encapsulating Security Payload (ESP [7]) and the IP Authentication Header (AH [6]), which offer confidentiality and authentication services. The Internet Key Exchange (IKE [4]) comprises protocols and mechanisms to automatically configure these IPSEC protocols and to maintain so-called "security associations." Once configured, IKE will set up and maintain IPSEC protected links autonomously as needed.

The downside is that configuring IKE and IPSEC is quite complex due to the flexibility needed to ensure inter-operability of different IKE and IPSEC implementations and to different security needs. Different encapsulation techniques, operation modes, and algorithms, which may vary for different network interfaces and destinations, making it difficult for system administrators to determine the specific security configuration used when communicating with a particular host.

Furthermore, emerging IPSEC management tools automatically configure VPNs by configuring IKE or – in case of manual keying – IPSEC to set up protected links between the gateways of the interconnected networks according to a company-wide security policy. Management tools are also needed to translate changes in the security policy into proper configuration changes of IKE and IPSEC. Our own work on the central management of IPSEC VPNs and the resolution of inter-operability problems of IPSEC implementations have underscored the need for additional tools that validate IPSEC configurations.

We tried several other ways to solve our problems: comparing round-trip times reported by the ping program to decide whether IPSEC encryption or authentication is actually applied and looking up IPSEC management commands of various IPSEC implementations to get unintelligible information about active "security associations." Eventually, we decided to develop our own tool with ease-of-use as a primary design goal. Our validation tool, called IPSECvalidate,

- validates what kind of encapsulation (ESP, AH, AH/ESP) and what mode of operation (transport, tunnel) is applied to IP packets from the local host to a particular remote host
- offers a command-line interface that can be used by other programs to validate VPNs consisting of multiple IPSEC links
- is independent of specific IPSEC implementations (as our scenarios span AIX machines, Windows2000 machines, Linux machines, and Cisco routers).

We will begin by describing the validation goals in more detail. Next, we will present the approach that we chose to validating IPSEC configurations. We will discuss alternative approaches and justify the chosen approach. Finally, we evaluate the potential impact of different kinds of attacks on the reliability of the validation tool's report. Specifically, we show that our tool is resistant against attacks from the network; i.e., attacks from the network cannot make our validation tool report that IPSEC protection is present although there is actually no protection.

Validation Goals

In this paper, we propose a methodology to test the validity of communication tunnels in an IP

network that have been set up using IP security protocols. IPSEC tunnels are used to construct a VPN over a public network such as the Internet. VPNs are typically used by multi-site organizations to interconnect their different locations. Until recently, this was accomplished by setting up dedicated tunnels such as Frame Relay circuits between two sites. However, a much less expensive solution is to use a common public network such as the Internet instead of dedicated interconnect links. This is accomplished by setting up IP based tunnels over the public IP network. At the same time, this raises a security issue since unlike dedicated Frame Relay circuits that carry an organization's traffic in an exclusive manner, the public IP network is a shared network. Thus, IPSEC mechanisms are used to set up secure tunnels that comprise a virtual private network amongst the different sites of a multi-site organization over a shared public IP network. This paper does not focus on the mechanism to set up such IPSEC tunnels but instead describes mechanisms to ensure that once such tunnels are set up, they operate as expected.

Our goal is to test whether the configuration of IKE and IPSEC on the various nodes of a VPN has been applied successfully. We consider a VPN to be comprised of a set of sites ("trusted domains") interconnected by an insecure Internet ("distrusted domain"), and want to ensure that the VPNs connecting these sites are correctly configured, thus providing protection. Furthermore, this needs to be verified before any of the client sites send data. Figure 1 depicts the basic scenario in which we did our work.

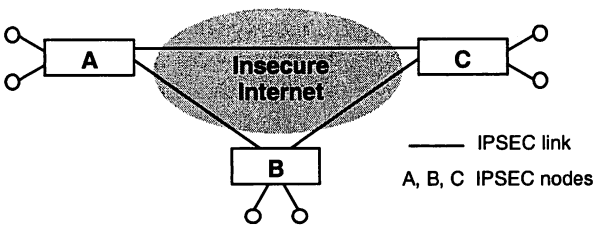


Figure 1: Basic VPN scenario using IPSEC.

This paper will present methodologies to test whether IPSEC associations have been successfully established between the three IPSEC nodes A, B and C of the VPN. These test methodologies are applied to each pair-wise connection, e.g., between A and B.

We assume that the IKE/IPSEC implementations on nodes A and B conform to the standards and that the connectivity configuration (e.g., VPN routing tables, policy tables) has been set up correctly. After nodes A and B have been configured with IKE/IPSEC, the problem is to check whether packets sent between A and B travel on the wire with the proper IPSEC parameters applied to the packets, e.g., whether they are sent as clear text or protected. IPSEC can be applied either through an Authentication Header (AH) or through an Encapsulating Security Payload (ESP), thus requiring two different sets of tests.

There are three possible methods we might chose to accomplish our goals: (a) snoop the packet on the sending node's network interface, (b) force an intermediate router on the path from A to B to send a portion of the packet back to the sending node, and (c) specially configure a router guaranteed to be on the path from A to B to return all packets to the sender if their header fits a specific pattern.

We implemented option (a) in our validation tool. We will discuss the alternative approaches and assumptions about attackers after presenting our implementation of this approach.

Note that the need for inspecting packets on the wire arises because the packets need to be checked for validity after IPSEC has been applied at the sending node and before IPSEC is applied at the receiving node. Otherwise, once a packet has traversed the IPSEC layers on both the sending and receiving nodes, there is no way to distinguish between packets that traversed the intervening network in clear text or with IPSEC applied.

Tool Description

The validation application is started by specifying a destination IP address and a protocol number:

```
$ipsecvalidate -d 192.168.19.48 -p 50
report:ICMP Packet Loss 0%
Transformations [IPSEC ESP tunnel mode]
are occurring as expected
$
```

IPSECvalidate tests whether all packets being sent to or received from destination address 192.168.19.48 are using the protocol specified by the -p option (e.g., protocol number 50, IPSEC ESP) for communication. To accomplish this, the application listens to all link layer frames both sent and received on all physical interfaces of the local host and examines all of these frames. It then compares the protocol number given as a command-line argument with the protocol field in the IP header of packets transported in these frames. The validation process consists of two parts:

- Any packet sent to or received from the remote host are inspected at the data link level regarding their encapsulation (protocol).
- ICMP Echo Request packets are sent to the destination host and ICMP Echo Reply messages are used to verify connectivity and to generate traffic, which is then inspected to verify the respective encapsulation.

The first step is to determine how many and what types of interfaces are present. This is typically accomplished by sending a query to the kernel. It is important to verify that each interface is operational and capable of hearing its own transmissions; flags in the data structure returned by the kernel will verify this. If any interface is not capable of hearing its own transmissions then it is not possible to verify the outbound communication.

The second step is to determine the source IP address to be used in the Echo Request packets. If a machine has a single interface then this interface's address is used. However, if a machine is multi-homed (which is the usual case for gateways) then the source address for the Echo Request packets has to be determined in order to recognize the corresponding Echo Reply packets while listening on the network interfaces. The source address to be used is determined by repetitive calls using the routing socket API. A routing socket is used to look-up the gateway for the remote host (destination). Once the gateway is determined the next step is to find the IP address of the interface that will be used to send the packets to the remote host (via the gateway). This also accomplished using the routing socket.

Once the list of valid interfaces has been created and the outgoing interface for the Echo Request packets has been determined, we create an appropriate number of slave threads, each listening to a single interface. Each slave thread will look for frames that match the source and destination IP addresses.

Several different methods can be used to listen to an interface. One method is to make use of a network tap; this is accomplished by creating a network tap socket and binding to a particular interface. The network tap receives copies of all packets that an interface sees. One problem encountered with the implementation based on AIX Unix was that only one application on a machine could have access to the network tap at one time. Therefore, if another application were currently using the network tap, the IPSECvalidate application would fail to run. The benefit of using the network tap is ease of programming.

We chose another method – snooping packets. On AIX we did this using the Berkeley Packet Filter API [9]; on Linux, we used the packet capture library (libpcap [2]).

Once the slave threads have been created and are listening to their respective interfaces, the main thread will send out a series of Echo Request (ping) packets to the destination host. The main thread also listens for Echo Reply packets to verify that the destination was reached and to check for packet loss. We include the process ID (PID) of the main thread in the Echo Request packets and inspect the PID payload of the received Echo Reply messages to make sure we count the replies to only our own requests.

We guard each local host interface by a listening thread because we might receive IP packets from the remote host via different interfaces. Each slave thread counts those filtered packets whose protocol field matches or doesn't match the protocol specified in the command line.

Once the main thread has received the ping responses or a time-out occurred, it will terminate the slave threads and release any resources associated with them. At this point the slave threads should have seen at least as many frames as Echo packets have been sent and received. If any slave thread received a frame where the source and destination IP addresses matched but the protocol field was not as specified then the transformations are considered to be not occurring as expected and the tunnel is not working correctly. Otherwise the transformations are occurring as expected.

If called with the "quiet" command line option, IPSECvalidate does not produce any output but communicates the validation results in the return value. This option can be used by other applications to dynamically determine the protection applied to IP packets exchanged with a particular remote network node, e.g., to support access control decisions.

The tool was implemented for AIX and Linux. On AIX, we used a modified packet capture library to capture and filter layer 2 frames and the standard routing socket to determine interfaces and routes. On Linux we used the standard packet capture library [2] and the library of IPRoute2 [8] to determine interfaces and routing information. The packet capture library supports numerous operating systems including FreeBSD, BSD, Linux, HP-UX, and Solaris. IPSECvalidate can be ported to other Unix-based operating systems by adapting the packet capture and routing socket calls as needed to the interfaces offered by the respective system.

Alternative Approaches

We considered several other approaches to the problem, only to find they were not feasible.

One approach we tried was to develop a protocol that runs between both of the machines in order to validate a secure tunnel. Using such a protocol, an initiating machine would send an initial probe message to a remote machine with which the IPSEC tunnel has been established. The other machine would then respond with a reply message which will only be generated if the corresponding packet was encrypted properly. If such a protocol can be developed, it will provide functionality which is similar to the one implemented by our tool.

However, because of the method in which IPSEC has been designed, the existence of a tunnel is transparent to applications running above the IP-layer. Therefore, an application layer implementation of this protocol would not be feasible. Such a protocol would need to be incorporated as a part of the IPSEC implementation. Since no such standard protocol is currently considered by the IPSEC working groups within the IETF, we would have to implement a non-standard extension to the protocol – something we did not find acceptable.

A second approach we considered was to use the TTL-expiration scheme used by programs such as traceroute. In this case, we would inspect packet headers after the relevant IPSEC transformations have been

applied. In order to capture packets after the IPSEC transformation, packets are created using a TTL field which is bound to expire, e.g., a TTL of 1. This would cause the next-hop router to send back the IP-header of the transformed packet and an additional eight bytes of the ESP/AH header. However, the difficulty lies in creating an IP packet with the TTL set to 1, which will then be sent through the IPSEC encryption routines. When IPSEC is used in the tunnel mode, the TTL of the outer header is often set to the IP default TTL, thus the packet will only be returned back by the network at the other end of the IPSEC tunnel, rather than from the next hop router. This scheme would not be able to validate the proper operation of IPSEC in the tunnel mode because the returned packets cross the insecure Internet unprotected. Even if the remote router uses IPSEC to protect such packets, they cannot be used to validate the IPSEC configuration as we would have to assume that IPSEC is working correctly for the validation – something we did not find acceptable

Security Evaluation

This section illustrates the benefits of the IPSEC validation tool. First, we interpret the output of IPSECvalidate in the regular case. Then, we discuss the interpretation of the output under different assumptions: We assume in turn insecure hosts, wrongly configured IKE and IPSEC policies and "security association" databases, and attackers having access to the network.

If both the local host (A) and the remote host (B) work correctly, then IPSECvalidate reports whether transformations (e.g., AH, ESP, AHESP) are occurring to data packets sent to and received from host B. If the reported ICMP packet loss is less than 100% then the IPSEC configurations of host A and host B are interoperable. Since the validation tool inspects all packets between A and B on all interfaces (on host A), route settings do not affect the validation result.

When examining the robustness of our tool against attackers, we restrict our discussion to the following scenarios:

- Local host A is insecure, i.e., its runtime environment does not work as expected.
- IKE/IPSEC configuration files do not reflect the users' security expectations.
- Attackers have access to the network and can read, replay, insert, and delete messages.

The first two scenarios involve compromised software and hardware (Trojan Horses [1]) or incorrect configuration, whereas the third scenario assumes external attackers. Figure 2 illustrates the validation tool's environment and points vulnerable to attacks.

This figure also shows how connectivity of hosts A and B is validated by the Echo protocol (which operates on top of the IPSEC sublayer). If IPSEC transforms ICMP packets the same way it transforms other IP packets, the IPSEC sublayers of hosts A and B are inter-operable for IP traffic if connectivity is reported. Other packet types, e.g., ARP packets, have mostly local significance and are usually not protected by IPSEC.

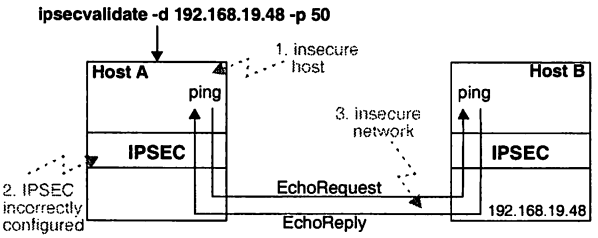


Figure 2: Points vulnerable to attacks.

If host A is corrupted due to internal attacks then users cannot securely interact with it; a validation tool running on this host is useless. If host B is corrupted, then using properly configured IPSEC to communicate with this host does not offer benefits; host B could distribute the actual session keys to the insecure Internet or leak any information through unprotected channels. The output of the validation tool is not useful in either case. This underscores the importance of secure runtime environments to reliably control and protect network access.

If the IPSEC configuration is corrupted on only one of the hosts, host A cannot communicate with host B. IPSECvalidate will report 100% ICMP loss in this case. Basically, 100% ICMP loss occurs if the network connection between hosts A and B is interrupted, ping is blocked by packet filters in between A or B, or hosts A and B have incompatible IKE or IPSEC configurations. If connectivity between hosts A and B over IPSEC is given then the IPSEC encapsulation is validated and the user can decide whether the configuration is as expected.

We will now examine how active attacks from inside the network can affect the output of IPSECvalidate. Such attacks – aimed at deceiving IPSEC protection – must generally be assumed when connected to the Internet.

Attackers cannot deceive that transformations are occurring, (i.e., they cannot provoke false positive transformation reports) because outbound packets are inspected before they enter the network; attackers cannot forge these packets from within the network. Because IPSECvalidate actively sends Echo Request packets, at least these outbound packets will be seen by one of the listening threads; these packets will reveal to the listening thread at the respective outbound interface whether expected transformations occur.

It is not possible for an external attackers to fool the tool into believing that there is connectivity, i.e., provoke false positive connectivity reports, if IPSEC uses authentication or encryption because the Echo

protocol is applied on top of IPSEC. IPSEC will discard inserted or replayed packets. If no replay detection is used, replay attacks are still difficult as we include the ID of the sending process in the ping packets and check it when receiving Echo Replies. This PID is likely to change for different invocations; hence even without authentication and replay protection, attackers cannot successfully replay Echo Reply messages of former connectivity checks (even if the IPSEC session key does not change). A random number can be added to the PID to achieve stronger protection against replay attacks.

Nevertheless, active attacks originating from the network (e.g., through replay, insertion, or deletion of messages) can provoke false negative reports:

- The tool can incorrectly report that transformations are not occurring as expected; this happens because IPSECvalidate does not interact with the IPSEC implementation. Therefore, it cannot validate the authenticity of incoming packets. Accordingly, attackers can insert forged packets or replay old packets that are not IPSEC-encapsulated; those are inspected by IPSECvalidate as incoming packets from host B.
- The tool can incorrectly report that connectivity is not given; this happens because attackers can selectively filter Echo Request or Reply packets exchanged between hosts A and B if no encryption is used.

In summary, IPSECvalidate can be used to verify that all IP packets between a pair of hosts are transformed using the expected security protocols and modes, regardless of the interfaces used to transmit and receive the packets.

Finally, IPSECvalidate cannot determine whether ESP actually uses strong encryption. Nevertheless, the validation tool does look for the IP header within the ESP body. If it finds the IP header at the expected offset then chances are that ESP does not use encryption (NULL encryption). However, if the IP header is not found we cannot conclude that strong encryption is used. Consequently, the tool is useful to determine the IPSEC encapsulation but it is at this time not thought to validate the theoretical strength of transformations (determined by algorithms and key lengths).

Summary & Outlook

In summary, IPSECvalidate can be used to verify that all IP packets between a pair of hosts are transformed using the expected security protocols and modes, regardless of the interfaces used to transmit and receive the packets. It also validates whether both hosts have compatible IKE and IPSEC configurations (connectivity over IPSEC). The validation tool is independent of the respective IKE and IPSEC implementations because it is exclusively based on standardized IPSEC protocol information and because it does not need any access to IPSEC databases. The tool assumes that the hosts are working properly and that the algorithms used within the transformations are applied correctly.

IPSECvalidate has been developed to support the validation of VPNS based on IPSEC tunnels. This can be achieved by running IPSECvalidate on all participants of a VPN and analyzing the results either on a central management node or locally on the VPN clients. The tool has proven to be very useful during the development of configuration tools for IPSEC-based VPNs. Local administrators and users will benefit from the tool because it makes normally transparent security mechanisms visible on demand.

IPSECvalidate can be used to verify any IP-based encapsulation protocol, e.g., GRE, IP-IP, or IPComp [11], simply by specifying the appropriate protocol number via the command-line options.

Future Work & Availability

Possible future extensions include heuristics that determine based on compression gain or code distribution with higher reliability whether ESP actually encrypts data or not.

We are going to release IPSECvalidate binaries for AIX and Linux to the community. Afterwards, we intend to go through the process to make the Linux source code available under GPL.

Acknowledgment

The authors would like to thank Adam S. Moskowitz for his very useful comments and suggestions, which considerably improved the presentation of this contribution.

Author Information

Arup Acharya received his B.Tech (Hons.) from IIT, Kharagpur and Ph.D. in Computer Science from Rutgers University in January, 1995. He was briefly associated with WINLAB (Wireless Information Networks Lab), Rutgers University as a post-doc working on the interaction of IP multicast with Mobile IP. He joined NEC Computers and Communications Research Labs, Princeton in May, 1995 where he worked on protocol architectures for high speed wired and mobile wireless networks. Arup joined IBM TJ Watson Research Center in Jan 2000 where his current work involves leveraging MPLS core networks to design a fast and scalable web switching infrastructure and architectures for pay-per-use Internet access through public WLAN and Bluetooth access points. Arup has been awarded three patents in high speed wired/wireless networking. He has published more than thirty papers in conferences and journals and has been a technical program committee member for MobiCom for the past three years. His homepage is http://www.research.ibm.com/people/a/arup/.

Mandis Beigi received her Bachelors of Engineering in Electrical Engineering from the State University of New York at Stony Brook in 1993. She

received her Masters of Science in the field of Electrical Engineering from Columbia University in 1995. She has been working at IBM since 1994. She currently works at the IBM T. J. Watson Research Center in the enterprise networking department and is also a Ph.D. student at Columbia University. She has worked on quality of service, service differentiation and network monitoring and management.

Raymond Jennings III is a research engineer at the IBM Watson Research Center. He obtained his MS in Computer Engineering from Manhattan College in 1996, and is currently pursuing his Ph.D. at Polytechnic University. His interests include operating systems and network performance.

Reiner Sailer is a Research Staff Member at the IBM T J Watson Research Center at Hawthorne, NY. He is an expert in network security, general purpose secure runtime environments, and security architectures for distributed applications. He holds a masters in Computer Science from the University of Karlsruhe and a doctorate of Electronic Engineering from the University of Stuttgart, Germany.

Dinesh Verma is a research manager at the IBM Watson Research Center. He obtained his Ph.D in Computer Networks from U. of California, Berkely in 1991, and has since worked at Philips Research and IBM Research in the topic of TCP/IP and ATM communication networks. He has authored three books and over thirty publications related to networking. His current interests include network performance and QoS, security, policy based management, content distribution networks and peer to peer networks.

References

[1] J. Anderson, "Computer Security Technology Planning Study," *ESD-TR-73-51, Vol I+II*, HQ Electronic Systems Division, Hanscom AFB, Ma., 1972.

[2] Network Research Group at the Lawrence Berkeley National Laboratory. *LIBPCAP 0.4: Packet Capture Library*, ftp.ee.lbl.gov, 1997.

[3] N. Doraswamy and D. Harkins, *IPSEC – The New Security Standard for the Internet, Intranets, and Virtual Private Networks*, Prentice Hall, 1999.

[4] D. Harkins and D. Carrel, *RFC2409 – The Internet Key Exchange (IKE)*. Proposed Standard, 1998.

[5] S. Kent and R. Atkinson, *RFC2401 – Security Architecture for the Internet Protocol,* Proposed Standard, 1998.

[6] S. Kent and R. Atkinson, *RFC2402 – IP Authentication Header*, Proposed Standard, 1998.

[7] S. Kent and R. Atkinson, *RFC2406 – IP Encapsulating Security Payload*, Proposed Standard, 1998.

[8] A. Kuznetsov, Institute of Nuclear Research, Moscow. "IPROUTE2," ftp.inr.ac.ru/ip-routing or ftp.sunset.se/pub/Linux/ip-routing, 1999.

[9] S. McCanne and Van Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," *USENIX Conference*, 1993.

[10] D. Piper, *RFC2407 – the Internet IP Security Domain of Interpretation for ISAKMP,* Proposed Standard, 1998.

[11] A. Shacham, R. Monsour, R. Pereira, and M. Thomas, *RFC2393 – IP Payload Compression Protocol (IPComp),* Proposed Standard, 1998.

ScanSSH – Scanning the Internet for SSH Servers

Niels Provos and Peter Honeyman – University of Michigan

ABSTRACT

SSH is a widely used application that provides secure remote login. It uses strong cryptography to provide authentication and confidentiality. The IETF SecSH working group is developing SSH v2, an improved SSH protocol that fixes cryptographic and design flaws in the SSH v1 protocol. SSH v2 compatible server software is widespread.

Recently discovered security flaws make it critically important to find vulnerable SSH servers and update them. In this paper, we describe a method to determine with good precision how many servers supporting the various protocol versions have been deployed on the net.

We describe the design and implementation of ScanSSH, a scanner that probes SSH servers for their software version, and discuss the results of scanning the Internet and our local networks for several months.

Introduction

SSH is a client/server application that provides secure remote login [6]. SSH uses strong cryptography to provide authentication and confidentiality. The IETF SecSH working group is developing SSH v2, an improved SSH protocol that fixes cryptographic and design flaws in the SSH v1 protocol.

Among all security sensitive services, access to SSH servers is widely unfiltered. SSH v2 compatible server software is widely available, yet recently discovered security flaws in SSH software make updating many older servers on the Internet critically important. In this paper, we describe a method to determine with good precision how many servers supporting the newer protocol version have been deployed on the net.

In particular, we designed and implemented a scanner that probes SSH servers and classifies them according to their advertised version number. ScanSSH is the result of this effort.

In this paper, we describe the design and implementation of ScanSSH and our experiences in scanning the Internet and our local nets for a period of several months. We illustrate changes in the deployment of SSH protocols as measured by our software.

ScanSSH supports fast scanning of large networks, so we have used it to classify random SSH servers on the Internet and thereby obtain a rough proportion of the fraction of SSH servers of various breeds.

Our Internet measurements scan over two million addresses per run on a regular basis. The scans can be distributed over several machines. It is possible to pick non-repeating random addresses from specified network ranges, as well as to exclude networks with alert and overly cautious administrators. We use ScanSSH on our internal network to find all SSH servers and to ensure that all vulnerable ones have been updated.

Coincidentally, during the development and testing of ScanSSH, a major security hole was discovered in most of the known versions of SSH. We describe our experiences in using ScanSSH as a tool to help local administrators update their SSH servers to address these vulnerabilities.

The remainder of this paper is organized as follows. In the first section, we present the design goals for ScanSSH. We present implementation details and a performance analysis in the next two sections. Subsequently, we discuss the results from probing the Internet and local university networks. We present related work in and conclude in the final two sections.

Design Goals

In this section, we describe the design goals that underly the development of ScanSSH.

The scanner has two questions to answer:

- What is the ratio of deployed SSH v2 servers to SSH v1 servers on the Internet?
- Which hosts on a network run vulnerable SSH servers?

To determine a server's protocol version, it suffices to look at the first message in the SSH protocol. The SSH v2 transport draft states that when a connection has been established, both sides must send an identification string of the form SSH-protoversion-softwareversion [5].

Servers that support the SSH v2 protocol use a protocol version of either 2.0 or 1.99. We use the information contained in the softwareversion field to determine whether a server is running a vulnerable server version.

To retrieve this information, it is necessary to establish a TCP connection to the destination host and read the first protocol message in the SSH protocol.

We can estimate SSH server deployment by randomly sampling hosts on the Internet. The distribution obtained from a random sample is close to the actual distribution, given that the random sample is large enough.

IPv4 can address about four billion addresses, so we want to be able to scan addresses quickly, even for a large sample. IPv6 is not widely deployed and the address space is too large for random sampling, so we ignore IPv6 addresses.

To find vulnerable SSH servers on a given network, we need to be able to scan specific networks. Inevitably, we run afoul of network administrators suspicious of our scans. For that reason, we need a way to exclude particular hosts or networks from a scan.

In summary, we need an efficient scanner that takes random or complete samples of a number of networks while excluding particular hosts that suspicious administrators do not want scanned.

Implementation

In this section, we discuss how the implementation of ScanSSH achieves the design goals from the previous section.

Producer-Consumer Model

ScanSSH implements a producer-consumer model [1]. The producer is implemented as a single process that discovers reachable hosts. The number of consumers is configurable. The producer feeds addresses of reachable hosts to the consumers. Once a consumer reads an address, it establishes a TCP connection to obtain the SSH version string. The result is returned to the central producer and printed to stdout.

Address Generation

Networks and hosts to be scanned are specified on the command line either as a single IP address or in classless inter-domain routing (CIDR) [2] notation.

The addresses can be modified with special prefixes that determine if the scan is distributed over more than one machine or if addresses are randomly generated.

The following example line specifies random address generation for the class B network 192.168 and the class C network 10.1.0:

```
random(0,Apollo)/(192.168.0.0/
                  16 10.1.0.0/24)
```

The zero specifies that all addresses should be scanned. Any other number specifies a limit on the number of hosts that should be scanned. The string "Apollo" is a seed for the pseudo-random number generator. Having a seed allows us to repeat a scan or distribute it on multiple hosts, because the generation of random numbers remains the same.

ScanSSH generates chunks of 64,000 addresses. The addresses in an address range are represented by a counter. The counter starts at zero and counts up to the number of available addresses. If the end of one network is reached, the next value of the counter represents the address at the beginning of the next network.

It is simple to generate addresses at random. Encrypting the counter with a block-cipher yields a random number that we can map to an Internet address. Using a counter guarantees unique addresses. Because encryption is bijective, the encrypted counter also produces unique addresses.

Most block ciphers use 64-bit blocks or larger. However, if we want to generate addresses for a /25 network, we need a block cipher using 7-bit blocks. We create a a variable block size that is related to the Tiny Encryption Algorithm (TEA) [4]. While the cryptographic security of TEA is not known with certainty, we don't rely on its security, just its bijectivity.

A single round of the variable block size cipher looks as follows:

```
sum += 0x9e3779b9
cnt ^= rndsbox[(cnt^sum) & sboxmask]
                          << kshift;
cnt += sum;
cnt = ((cnt << left) | cnt >> right))
                             & mask;
```

We use the following constants:

```
kshift = left = bits / 2;
right = bits - left;
sboxmask = (1 << kshift) - 1;
```

The input to the block cipher is the block size bits and a counter cnt. After iterating the round 32 times, cnt contains the encrypted result. It is easy to see that the encryption function is bijective by observing that each round is reversible.

Producing Addresses

The producer process takes an address from the generated chunk of addresses and sends a TCP SYN packets to the host specified by the address. The destination port is set to 22, for SSH.

In the default configuration, ScanSSH supports 4096 outstanding TCP SYN packets. If we do not receive a reply in a certain time period, the TCP SYN packet is resent and the timeout increased. This process continues until the retry limit has been received. In that case, ScanSSH reports a timeout.

If we receive a response from a host, it is either a TCP RST segment or a TCP SYN/ACK segment. In the former case, ScanSSH reports a refused connection, and in the latter case, the address is put on a queue from which the consumer processes can feed.

Consuming Addresses

The consumers feed from the address queue in a round-robin fashion. We use standard inter-process communication to send an address to a consumer process. The consumer then establishes a TCP connection

to the remote host, and waits for the SSH identification string.

After the identification string has been received, it is printed to stdout and the consumer sends its own version string SSH-1.0-SSH_Version_Mapper to the SSH server. The protocol number 1.0 causes a SSH server to close the connection, because protocol 1.0 is not specified.

Address Exclusion

ScanSSH reads an exclusion file that specifies the hosts that should not be scanned. If an address that falls in an excluded network range is generated, the address is ignored. In addition, private and multicast networks are ignored by default.

Performance Analysis

The performance of ScanSSH depends on the percentage of responsive hosts. By default, ScanSSH manages a list of 4,096 outstanding TCP SYN packets. When ScanSSH receives a reply to a SYN packet, the corresponding address is removed from the list and replaced by a new address. However, if a host does not respond to any packets, its address entry remains on the list for about half a minute. That means that the unresponsive hosts are the limiting factor for the scan rate.

Scanning the CITI network consisting of 384 addresses connected via a 100 MBit network takes about 36 seconds. The scan reports 62 active hosts with 44 of them running SSH servers. With a primed ARP cache, scanning the active hosts takes 0.2 seconds. Scanning only the 322 unresponsive hosts takes 36 seconds.

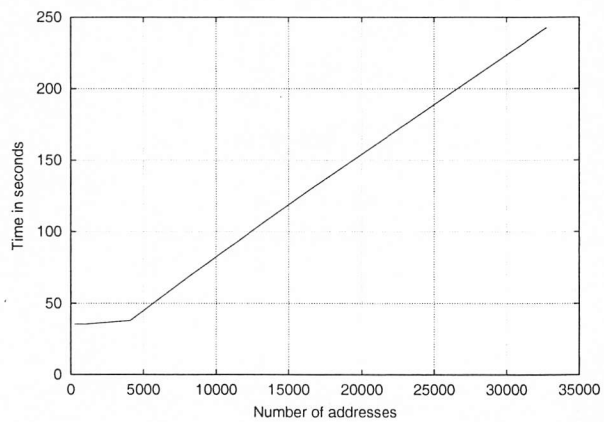


Figure 1: Worst case performance of ScanSSH when scanning only unresponsive hosts.

We measure the worst case performance of ScanSSH by scanning unroutable networks; see Figure 1. The average scan rate is about 135 hosts per second. We expect the worst case behavior to be a good estimate for scanning random addresses on the Internet, as most are unresponsive.

Measurement Results

In this section, we present results from measuring the deployment of SSH servers on the Internet and from assessing the number of vulnerable SSH servers running at the University of Michigan.

SSH Server Deployment

We have been scanning over two million random addresses on the Internet since July 2000. The scans are repeated monthly.

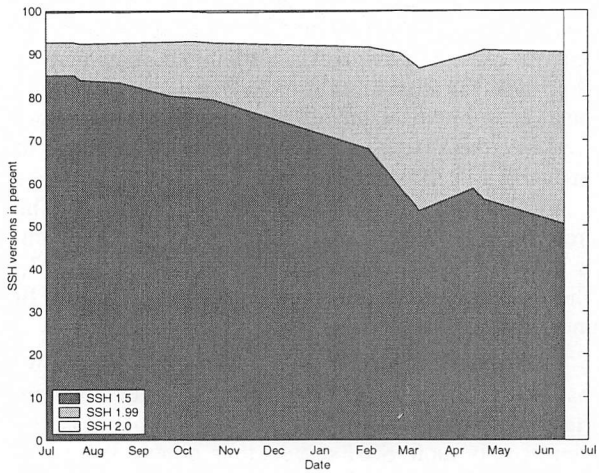


Figure 2: Deployment of SSH protocol versions on the Internet.

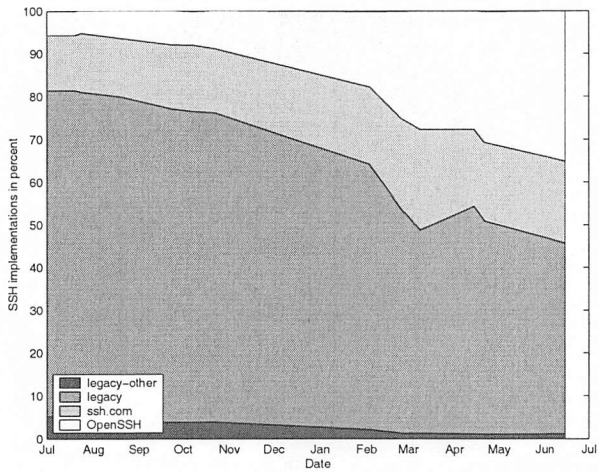


Figure 3: Deployment of different SSH server software on the Internet.

Figure 2 shows the deployed SSH protocols from July 2000 until the end of June in 2001. In July 2000, about 15% of all SSH servers supported SSH v2. At the end of our measurements in June 2001, almost 50% of all SSH servers support the version 2 protocol. Interestingly, the fraction of servers that support only SSH v2 remains almost constant. They change from 7% to 10%, whereas the percentage of servers that support both SSH v1 and SSH v2 increases from 8% to 40%. Examining Figure 3 provides an explanation

for this behavior. The increase of SSH v2 capable servers is mostly due to OpenSSH. OpenSSH's contribution of SSH v2 servers increased from 1.7% in July 2000 to 30% in June 2001.

Because we sample hosts randomly, the data can be used to estimate the number of responsive hosts on the Internet, *i.e.,* hosts that are connected to the Internet and not hidden behind a firewall. Given the number of responsive hosts, we can estimate the percentage of hosts that run SSH services.

The upper graph in Figure 4 shows the data as measured. There is no significant change in the percentage of responsive hosts during our scan period. The slight variations might result from changes in the location from where the scans were conducted. However, we notice stronger variations in the percentage of hosts that run SSH servers. In February 2001, there is a significant drop in hosts running SSH servers. The drop might be correlated to a serious security problem in the "CRC32 Compensation Attack Detector" [7]. The thicker line in the graph represents a linear fit under that assumption.

However, a closer examination of the data shows that the distribution of SSH servers is not uniform. Figure 5 shows that certain areas in the network run significantly more SSH servers than others. We notice a strong decrease of servers for these addresses between our scans. We observe a similar decrease in the number of responsive hosts. One possible explanation is that the address of our scanning hosts has been filtered. When we remove the part of the network in which we measured the drop in responsive hosts, the number of SSH servers increases linearly, as shown in the lower graph of Figure 4.

SSH Vulnerability Scanning

Because SSH servers are a critical component of the Internet infrastructure, it is important to react quickly to security holes discovered in SSH server software. The remote root hole described in the previous section was a motivation for us to help to update vulnerable SSH servers at the University of Michigan.

To find vulnerable SSH servers, we scanned about 400,000 addresses on a daily basis. We found about 2,300 hosts running SSH services. To identify if a host was vulnerable, we check the software version returned by the SSH server against a table of software known to be vulnerable. We assign the category "unknown" to hosts for which we have no vulnerability information. The results are shown in Figure 6.

In February 2001, we found that 90% of the SSH servers running at the University of Michigan had known security problems. In the course of our research, we sent lists of vulnerable hosts to the administrators of the respective networks. After two months, the percentage of vulnerable SSH servers went below 20%. As security researchers, we think of

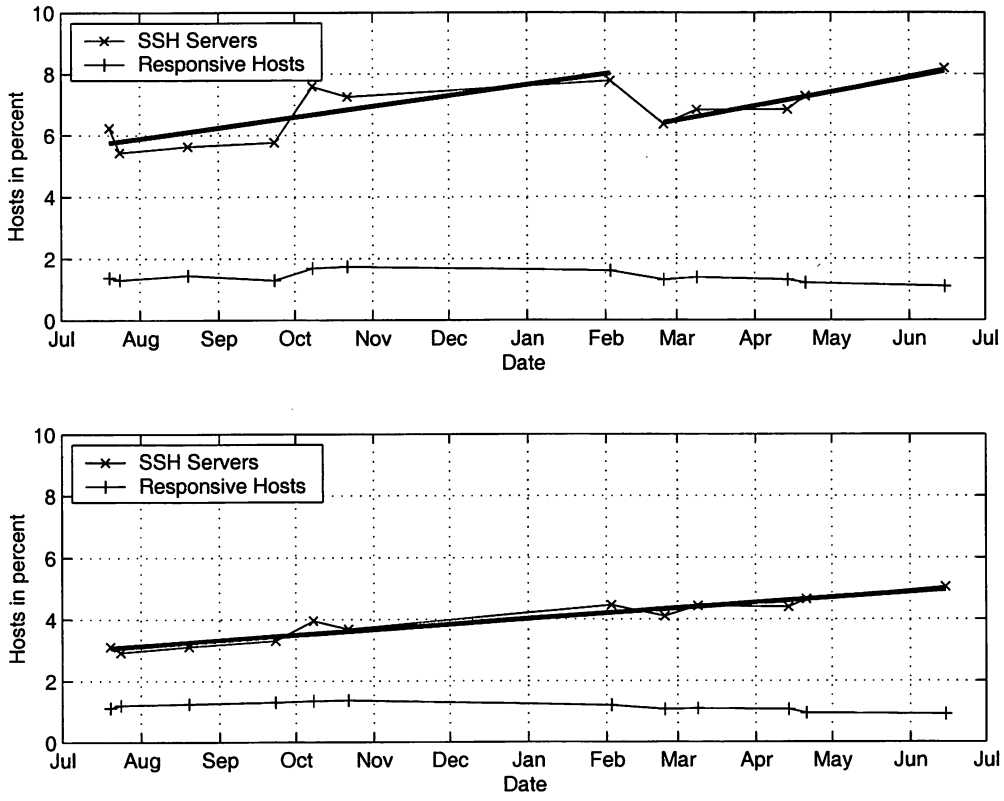


Figure 4: Percentage of responsive hosts and percentage of hosts running SSH servers. The upper graph shows the data as measured, the lower graph shows adjusted data.

this as a long time period. However, given the administrative needs and structure of the university (or any bureaucracy), the reaction time is not surprising.

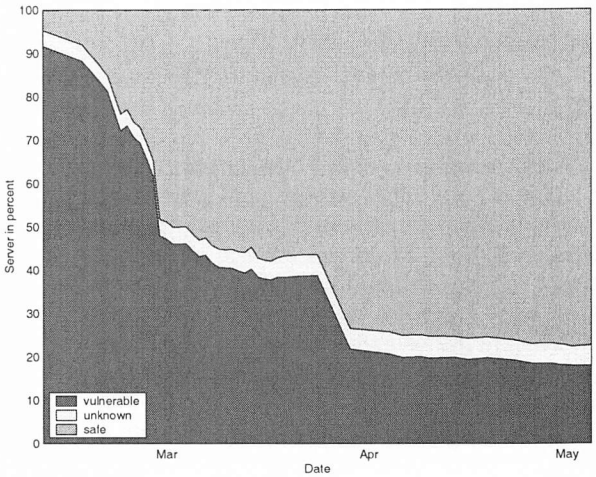


Figure 6: Percentage of SSH servers that are either safe or vulnerable to known security holes.

Related Work

Nmap is a port scanner that allows scanning multiple hosts at once [3]. Nmap scans hosts sequentially, i.e., a port scan on one host must complete before the next host is scanned. While that might be adequate for small networks, it is too slow for scanning many addresses. ScanSSH scans hosts in parallel and thus achieves higher scanning speeds.

Conclusion

We described the problem of measuring the deployment of the SSH v2 protocol and finding vulnerable SSH software.

To solve these problems, we implemented an efficient SSH scanner. We reasoned that efficiency was our main design goal and showed how our implementation meets it.

We measured the deployment of SSH servers on the Internet and showed how ScanSSH has been used to update vulnerable servers to more secure software.

ScanSSH has been released as UNIX source code under a BSD license and is available at http://www.monkey.org/provos/scanssh/ .

Acknowledgments

We thank Bob Beck, the University of Alberta, CORE-SDI and Peter Galbavy for providing network and computing resources for our scans. We thank David Wagner for helpful conversations about random number generation and Theo de Raadt for reviews and helpful suggestions about data analysis. We also thank Adam Moskowitz for shepherding this paper.

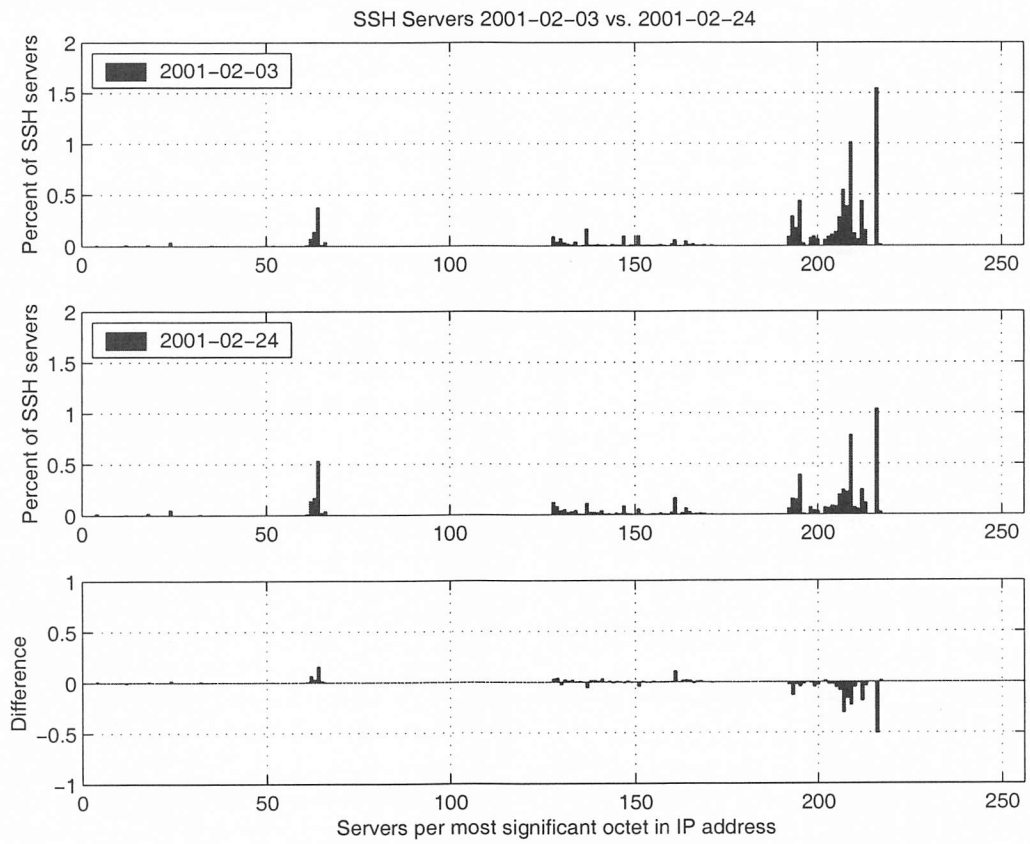


Figure 5: Comparison of SSH server distribution at the beginning and end of February 2001.

References

[1] Andrews, G. R. and F. B. Schneider, "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys*, Vol. 15, Num. 1, pp. 3-43, March, 1983.

[2] Fuller, S., T. Li, J. Yu, and K. Varadhan, "Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy," *RFC 1519*, September, 1993.

[3] Fyodor, "Remote OS Detection via TCP/IP Stack Fingerprinting," http://www.nmap.org/nmap/nmap-fingerprinting-article.html , October, 1998.

[4] Wheeler, D. and R. Needham, "A Tiny Encryption Algorithm," *Technical Report 355*, University of Cambridge, December, 1994.

[5] Ylönen, T., T. Kivinen, M. Saarinen, T. Rinne, and S. Lethinen, *SSH Transport Layer Protocol,* Internet Draft, work in progress, January, 2001.

[6] Ylönen, Tatu, "SSH – Secure Login Connections over the Internet," *Proceedings of the Sixth USENIX Security Symposium*, pp. 37-42, July, 1996.

[7] Zalewski, Michael, "Remote Vulnerability in SSH Daemon crc32 Compensation Attack Detector," *RAZOR Bindview Advisory CAN-2001-0144*, Feburary, 2001.

A Probabilistic Approach to Estimating Computer System Reliability

Robert Apthorpe – Excite@Home, Inc.

ABSTRACT

Probabilistic Risk Assessment (PRA) is a method of estimating system reliability by combining logic models of the ways systems can fail with numerical failure rates. One postulates a failure state and systematically decomposes this state into a combination of more basic events through a process known as Fault Tree Analysis (FTA). Failure rates are derived from vendor specifications, historical trends, on-call reports, and many other sources. FTA has been used for decades in the defense, aerospace, and nuclear power industries to manage risk and increase reliability of complex engineering systems. Combining FTA with event tree analysis (ETA), one can associate failure probabilities with consequences to clearly communicate risk both pictorially and numerically. Basic PRA techniques can help increase the reliability and security of computer systems.

Introduction

As system administrators, our primary responsibility is to maximize the availability of the systems in our charge. This responsibility is a constant challenge, exacerbated by economic and competitive pressure, the continuing rapid growth of the Internet, and institutions' increasing reliance on network services. As systems become more ubiquitous and more important to an organization, more emphasis is placed on reliability [1].

Often we desire a quantitative measure of reliability. We are inundated with diagnostic information and in recent years have focused on data analysis. Unfortunately, this data is only a historical record; it tells us little about underlying system behavior, dependencies, or latent vulnerabilities, and its predictive value is limited.

The computer industry lags behind other industries in terms of risk assessment methodologies. We rarely assess risk formally, usually only when designing a system. System administrators lack the most basic means of comprehensively analyzing and quantifying risks to the systems they maintain.

We seek a formal approach to risk assessment that offers a qualitative understanding of system behavior and vulnerability. The approach should be systematic and comprehensive, producing quantitative measures of risk and component importance consistent with observed historical data. We want a technique that produces lasting benefits for the effort spent and that can be understood and applied by the average system designer or administrator. Above all, we desire a method with an established record of success.

One such technique is Probabilistic Risk Assessment (PRA), a systematic method of enumerating the ways a system can fail. PRA considers both probability and consequences of individual events, affords a scalable level of detail, and can model human reliability as well as software and hardware reliability. Developed in the 1960s, PRA has seen widespread use in the aerospace, defense, and nuclear power industries [11].

Intent

This paper serves as a brief tutorial on event tree analysis (ETA) and fault tree analysis (FTA). I will model a typical mail receipt system using ETA, then decompose one event into a fault tree. I will quantify the fault tree and show metrics that help analysts estimate the importance of components. Finally, I will discuss limitations of the method and suggest areas for future study and research.

The techniques I describe have developed over the last 40 years and combine such fields as probability theory, graph theory, systems engineering, operations research, and statistics [19] [20]. Despite the age of technique, introductory material on PRA is scarce. Often PRA techniques are only taught at the graduate level making the subject even less accessible, especially to practicing technologists.

Reliability analysts must thoroughly understand the systems they model from the high-level systemic scale down to the component level. They also need broad knowledge of the interrelationships between systems and the tenacity to ferret out subtle dependencies. System administrators already have many of these skills and all that is needed is an introduction to the technique. PRA reduces risk assessment to a rational, mostly mechanical process.

Preliminary Analysis

Most formal risk or reliability assessments start with these three steps:

1. Identify the hazard
2. Identify relevant systems, components, and individuals
3. Bound the analysis

While this may seem simplistic and obvious, it's vitally important to know at the outset which events one is concerned with, which systems are to be analyzed, and the level of detail one will consider.

Example: Inbound Mail Transport

An organization (wazmo.org) considers it vital that no inbound mail is ever irrevocably lost. Reviewing the mail transport process [14, 15, 16, 17] we see the following relevant systems, components, and individuals:

- Remote sender
- Remote user host
- Remote MUA (mail user agent, e.g., client software)
- Remote MTA (mail transport agent, e.g., mail server software)
- Remote network
- The Domain Name System (DNS)
- The Internet
- wazmo.org's ISP
- Local network
- Local MTAs
- Local user host
- Local MUA
- Local recipient

Expanding our scope, we can consider the following:

- Power transmission and distribution network
- On-site environment (temperature, humidity, etc.)
- Off-site environment (likelihood of fire, flood, ice storm, earthquake, etc.)
- Human activity (inept or malicious)
- Extraterrestrial activity (solar flares, magnetic storms, etc.)
- Political climate (civil unrest, war)

As esoteric as some of these items seem, there's ample anecdotal and documentary evidence of systems being challenged by each of them [12] [13]. Emergency response organizations often rely on pagers which in turn rely on satellite systems; the reliability of pager networks is frequently taken for granted despite occasional outages due to satellite misconfiguration and solar flares. The same is true for systems such as the national electric power grid and the public switched telephone network.

To simplify our analysis we will only model portions of wazmo.org's local site, their ISP (mail servers, name servers), and the Internet (the connection from wazmo.org to cynistar.net, its ISP.)

The mail servers are listed in DNS (db.wazmo.org) in decreasing preference as follows:

```
wazmo.org 1D IN MX 10 shaft.mx.wazmo.org.
          1D IN MX 20 dolomite.mx.wazmo.org.
          1D IN MX 30 mta00.cynistar.net.
```

This shows two internal mail servers (shaft and dolomite) and an external backup mail server (mta00.) Shaft is the primary with a preference of 10, dolomite is secondary with a preference of 20, and mta00 is tertiary with a preference of 30.

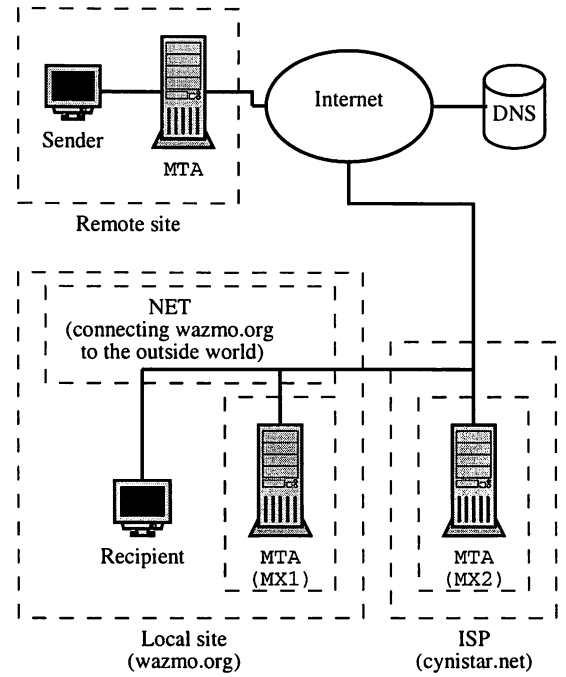


Figure 1: Simple mail transport model.

Fundamentals of Event Tree Analysis

An event tree is a diagram of all the events that can occur in a system, a graphical form of truth table. Event tree analysis is an inductive approach, in that it answers the question "What if event X happens?" One starts with an initiating event (root node) on the left side of the diagram proceeding though a series of branch points that represent the occurrence and magnitude of particular events. Each of the branches has an associated probability or split fraction. The end states (leaf nodes) represent the combination of events leading to a particular consequence. The result is a tree structure that clearly shows the events leading to particular consequences and allows a quantitative measure of risk to be derived without much effort. This technique is best illustrated with an example.

Event Tree Construction

Start by defining the consequence of interest and systems or functions relating to that consequence. In this case, we define the consequence as "loss of inbound mail" and we choose the systems to be:

- **DNS**: can the domain name system resolve the appropriate destination for inbound mail (specifically MX records)?
- **NET**: can remote hosts reach local mail servers? Can local users reach local mail servers?
- **MX1**: are local (primary) mail servers accepting mail?
- **MX2**: are remote (secondary) mail servers accepting mail?

Our initiating event is "mail sent to wazmo.org." This is a high-probability event for all but the smallest domains. Combining these events, we construct a *basic* event tree, one containing all possible branches. Some states make no sense and can be eliminated, leading to a *reduced* event tree. For example, if DNS cannot resolve the address of the receiving MTA (either MX1 or MX2), the states of the local network, and primary and secondary mail servers are irrelevant. If DNS, NET, and MX1 are available, the state of MX2 is irrelevant since mail will be delivered to MX1 and there will be no demand for MX2. Finally, if the local network (NET) is unavailable, the state of MX1 is irrelevant; once the connection attempt to MX1 fails the sending MTA will try MX2 next. The reduced event tree tersely and clearly explains a fairly complex system.

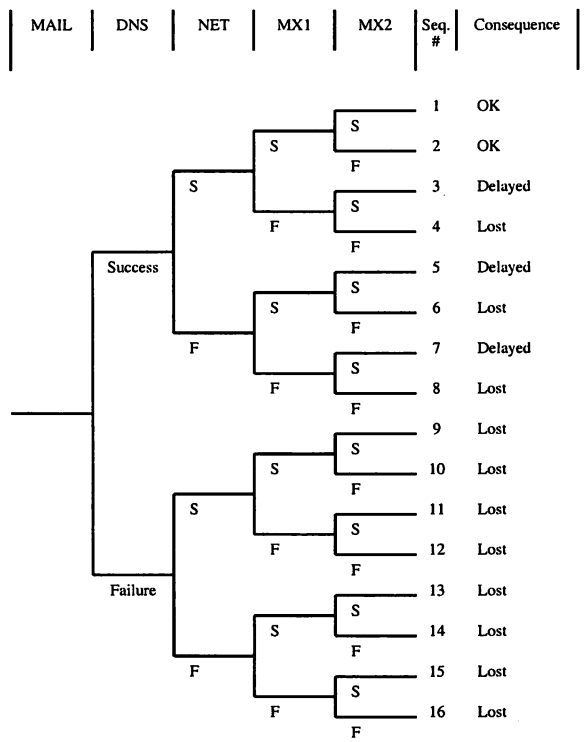


Figure 2: Basic event tree.

We can estimate reliability with an event tree by associating probabilities with each branch point. For example, we could send test messages to MX1 and MX2 and record the number of failures to estimate the reliability of the mail exchangers. If the reliability of DNS is given as P_{DNS}, reliability of the site's network and internet connectivity (NET) is P_{NET}, and reliability of MX1 and MX2 are given by P_{MX1} and P_{MX2} respectively, the probability of prompt mail delivery (sequence #1 in Figure 3) is

$$\begin{aligned} P_{ok} &= P_1 \\ &= P_{MAIL} \times P_{DNS} \times P_{NET} \times P_{MX1} \times \\ &\qquad (P_{MX2} + (1 - P_{MX2})) \\ &= P_{MAIL} \times P_{DNS} \times P_{NET} \times P_{MX1} \end{aligned}$$

where P_{MAIL} is the probability that inbound mail will arrive during the interval of interest. The probability of mail being delayed (assuming a delay in transport from MX2 to MX1 when it is eventually repaired) is the sum of the probability of sequences #2 and #4 from Figure 3:

$$\begin{aligned} P_2 &= P_{MAIL} \times P_{DNS} \times P_{NET} \times (1 - P_{MX1}) \times P_{MX2} \\ P_4 &= P_{MAIL} \times P_{DNS} \times (1 - P_{NET}) \times P_{MX2} \\ P_{delayed} &= P_2 + P_4 \\ &= P_{MAIL} \times P_{DNS} \times P_{MX2} \times (1 - P_{NET} \times P_{MX1}) \end{aligned}$$

And finally,

$$\begin{aligned} P_{LOST} &= P_3 + P_5 + P_6 \\ &= (1 - (P_1 + P_2 + P_4)) \\ &= (1 - P_{ok} - P_{delayed}) \end{aligned}$$

In this simple case each branch point has only two states {Success, Failure}; it is possible to have more than two states per event. For example, we could add a third state to MX1 and MX2 – {Success, Degraded, Failure} – and model another situation in which inbound mail is delayed. The sum of each state's probability at a branch point must add up to unity to encompass all possible events (e.g., DNS either works or it doesn't; MX1 is either fully-functional, degraded, or failed; there are no other states than the ones we consider.) Also, the probabilities do not need to be consistent along different paths, that is, P_{MX2} in sequences #2 and #3 need not have the same value as P_{MX2} in sequences #4 and #5. However P_{MX2} must be consistent in sequences #2 and #3 since by definition $P_{MX2_{success}} + P_{MX2_{failure}} \equiv 1$ at each branch point.

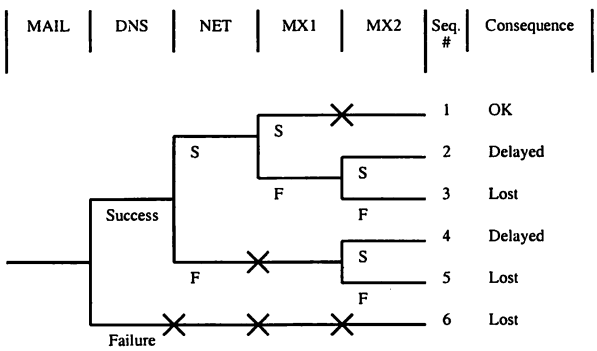


Figure 3: Reduced event tree.

The next task is to derive numerical values for each of these failure probabilities. One may estimate probabilities or use observed data but in more sophisticated models these values are derived from fault tree analyses.

Fundamentals of Fault Tree Analysis

The most common PRA technique is fault tree analysis (FTA). A fault tree is a logical model of the various parallel and sequential combinations of faults that will result in the occurrence of a predefined

undesired event. This event (known as a *top event* due to its position in the tree) is linked to more basic events through a number of intermediate events and logic gates detailed below. Fault tree analysis is a deductive technique which asks "How can event X occur?"

Fault trees provide a detailed graphical explanation of system behavior. In this respect, fault trees are a communication tool that can effectively explain complex systems to those not intimately familiar with the details of a system's design.

Concepts and Definitions

Most of this information is summarized from NUREG-0492 [2] and [19, 20]. It has been reordered somewhat for clarity.

Faults vs. Failures

For modeling purposes, we distinguish between failures (basic abnormal occurrences) and faults (higher-order undesirable events.) Component failures are a subset of a larger class of events known as faults. This helps us distinguish between more concrete basic events and more abstract intermediate events. The reason for this distinction will become clear later, especially in the case of command faults.

***State-of** Fault Categories*

Faults may be categorized as *state-of-component* faults or *state-of-system* faults. State-of-component faults suggest the fault should be decomposed into primary, secondary, and command faults. State-of-system faults are often caused by action outside a component and suggest that further decomposition of the fault is necessary. The decomposition process for state-of-system faults is not as mechanical as that for state-of-component faults.

Component Fault Categories

Component faults can be classified as primary, secondary, and command faults.

Primary faults are faults of a component that occur when the component is operating within its design specification, e.g., a web server rated at 50 TPS (transactions per second) fails at a load of 30 TPS.

Secondary faults are faults of a component that occur when the component is operating outside its design specification, e.g., a web server rated at 50 TPS (transactions per second) fails at a load of 90 TPS.

Command faults result when a component properly performs its intended function but at the wrong time or place, e.g., the fire suppression system in a data center discharges due to a spurious or premature signal from a detector or controller.

Events

Events can be categorized as *top*, *intermediate*, or *primary*, depending on their position in the fault tree. In computational tree terminology, primary events are leaf nodes and top events are root nodes. Primary events are further subdivided into *basic*, *undeveloped*, *conditioning*, and *external* events. Basic events are are considered atomic and are not expanded further. These are usually primary, secondary, or command faults. Undeveloped events are just that; they indicate events that are not considered atomic but for reasons of uncertainty or simplicity are not expanded further. Conditioning events show any conditions or restrictions on that apply to any logic gate (usually inhibit or priority-and gates.) External events (sometimes known as *house* events) signify events that are normally expected to occur; they are not faults by themselves. They are often used for sensitivity analysis. Conditioning and external events are rarely used in practice. See Figure 4 for a summary of event types and their symbols.

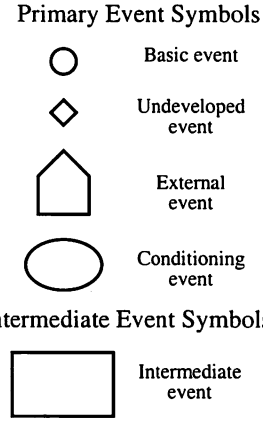


Figure 4: Event types.

Gates

The logic gates used in fault tree analysis include the familiar and, or, and exclusive-or gates. transfer-in and transfer-out gates are conveniences that allow a large fault tree to be broken into sections that fit on the printed page. The inhibit gate is a special form of the and gate composed of a single input event and a conditioning event. The priority-and gate is a special form of and gate where the input events must occur in a specified order (by convention input events must occur in order from left to right in the diagram.) The m-of-n gate is true when at least m of the n input events occur; this gate is used to model voting logic or multiply-redundant systems. Many of these gates are for convenience purposes. In practice, one can effectively model systems using only and and or gates, using transfer gates to break up the tree into printable sections. Figure 5 summarizes these gates and their symbols.

Cutsets

A *cutset* is a set of basic events which, if they occur, guarantees the top event will occur. A *minimal cutset* is a cutset which, if any event is removed, will no longer be a cutset (e.g., no longer guarantees the top event will occur.) The primary purpose of fault tree analysis is to identify minimal cutsets.

For the curious, "cutset" is a term from graph theory. Fault tree analysis works in *failure space*

looking for sets of events that prevent successful operation. It is possible to convert a fault tree to a connected graph of success events (events that must occur to ensure successful operation.) The set of events that disconnect (cut) a connected graph is a cutset [21]; this implies that fault tree analysis may have its origin in success path analysis (in graph theory parlance, the fault tree is the dual of a graph of success paths.) While it's far beyond the scope of this paper to explore etymology, history, and graph theory, it may be helpful to review the underlying mathematics and history for additional insight.

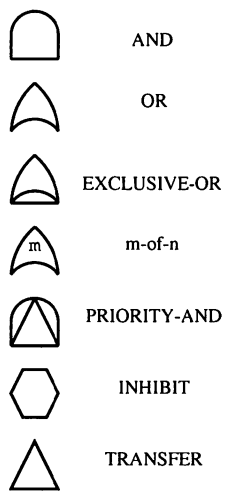


Figure 5: Gate types.

Fault Tree Construction

The basic steps in performing a fault tree analysis are

1. Define failures of interest (top events), usually by an inductive process such as event tree analysis. The combination of fault- and event-tree analysis is sometimes known as cause-consequence analysis (CCA).
2. Define the systems to be analyzed and the limit of resolution of the analysis.
3. Build logical models (fault trees) of the events that lead to each of the top events defined in Step 1.
4. Evaluate the models to determine the sets of basic events (minimal cutsets) that lead to each top event.
5. Optionally, quantify the likelihood of each minimal cutset using component failure probabilities.

To explore this technique, we return to our mail system analysis. For simplicity, we will only model the MX1 event show in Figure 3.

Fault Tree Analysis of Inbound Mail Transport System

We begin the analysis by defining our top event as "Local MTA fails to accept mail" This fault is clear, direct, and unambiguous. It is best to phrase events tersely and directly, usually in a single sentence containing no more than fifteen to twenty words. Remember, the top event states what happens; the fault tree explains how.

Now we define our system. We have two mail servers and a single dedicated switch in common as shown in Figure 6.

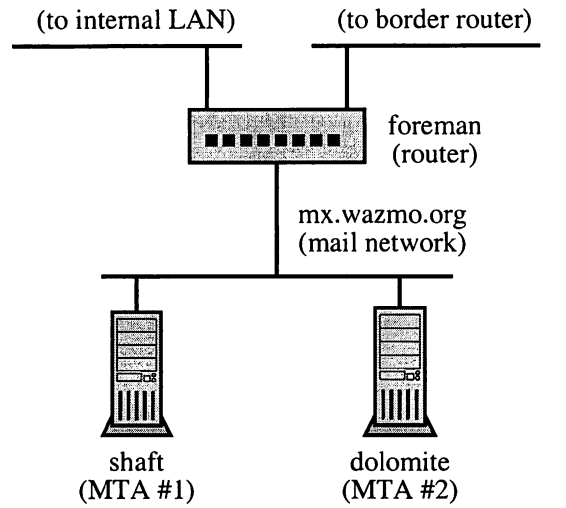


Figure 6: Physical description of local Mail exchangers (MX1).

The system operates as follows:

1. Incoming mail traffic is routed from the site's border routers to the incoming mail network
2. According to DNS, MTA #1 (shaft.mx.wazmo.org) is listed with a preference of 10, MTA #2 (dolomite.mx.wazmo.org) has a preference is 20, and the off-site mail server (mta00.cynistar.net) is listed with a preference of 30.
3. Remote mail servers will first attempt delivery to MTA #1. If MTA #1 will not accept incoming mail, the remote server will try to deliver mail to MTA #2. Failing that, it will then deliver mail to the off-site mail server. Undelivered mail is silently discarded.

Note that we assume no failures in the border routers and internal networking (NET), the off-site mail server (MX2) or in DNS. We only concentrate on the system we've defined as MX1; failures in these other systems will be detected when these systems are analyzed separately. One benefit of fault tree analysis is that we can decompose our analysis into smaller, more manageable pieces.

We can simplify the analysis by making some assumptions about the components and the system:

1. The mail servers are identical and are capable of handling all traffic directed to them under normal circumstances. That is, only one mail server needs to be available for the system to perform within its design specification.

2. Failures are permanent. No automated or manual recovery actions are assumed.
3. Components are independent of each other. This is a very important assumption and will be discussed later.
4. The router and both mail servers are served by the same source of electrical power.
5. We only consider faults in the components under analysis. We do not consider faults due to problems with cabling or room cooling, nor do we consider events such as theft, fire, flood, seismicity, etc.

The definition of a system includes modeling assumptions and design basis assumptions, the "design basis" being the range of conditions within which a system is designed to operate.

Next, we construct a fault tree using the rules described in the Fault Tree Handbook [2]:

- Ground Rule I: Write the statements that are entered in the event boxes as faults; state precisely what the fault is and when it occurs.
- Ground Rule II: If the answer to the question "Can this event consist of a component failure?" is "Yes," classify the event as a "state-of-component fault." If the answer is "No," classify the event as a "state-of-system fault." If the fault event is classified as "state-of-component," add an or-gate below the event and look for primary, secondary and command modes. If the fault event is classified as "state-of-system," look for the minimum necessary and sufficient immediate cause or causes. A "state-of-system" fault event may require an and-gate, an or-gate, and inhibit-gate, or possibly no gate at all. As a general rule, when energy originates from a point outside the component, the event may be classified as "state-of-system."
- No Miracles Rule: If the normal functioning of a component propagates a fault sequence, then it is assumed that the component functions normally.
- Complete-the-Gate Rule: All inputs to a particular gate should be completely defined before further analysis of any one of them is undertaken.[1]
- No Gate-to-Gate Rule: Gate inputs should be properly defined fault trees, and gates should not be connected directly to other gates.

The phrase "immediate, necessary, and sufficient" from Ground Rule II requires some clarification. We must clearly describe the current state of the system. Example: a computer is powered through an uninterruptible power supply (UPS) that holds 20 minutes of reserve power. If AC power is lost and does not recover before the UPS battery drains, the device fails, and the failure event is written as "Loss of AC power," not "Loss of power for more than 20 minutes." The *immediate cause* of the event is loss of power. The loss of power is *sufficient* to cause the event. If the computer has multiple redundant power supplies, an immediate cause of failure would still be "Loss of AC power,"; when that event is expanded, the *necessary* and *immediate* cause is "Loss of power from UPS 1 and 2."

Some analysts have compiled an additional list of heuristics to simplify fault tree construction [19].

- Replace an abstract event by a less abstract event.
- Classify an event into more elementary events.
- Identify distinct causes for an event.
- Couple trigger event with "no protective action."
- Find cooperative causes for an event.
- Pinpoint a component failure event.

We start by expanding the top event E1 "Local MTA fails to accept mail." E1 is a *state-of-system* fault so we look for its necessary, immediate, and sufficient causes. We define event E2 as "Router unavailable" and E3 as "Inbound mail service unavailable" and combine them into and or-gate. Our tree is now E1 = (E2 or E3). See Figure 7.

Note transfer gates 1 and 2, leading to Figures 8 and 9, respectively. This leads to a bit of jumping between figures as we follow the Complete-the-Gate Rule.

Event E2 "Router unavailable" is a *state-of-system* fault so we again look for necessary, immediate, and sufficient causes. We expand E2 into two events, E4 "Router failed" and E5 "Router OOS[2] for maintenance." Note we are modeling human action and operations procedure as well as random failure. See Figure 8.

Event E3 "Inbound mail service unavailable" is a *state-of-system* fault; we model this as E6 "No MTA available" and E7 "Common-cause failure of all MTAs" both feeding into an or-gate. Common-cause failure is a special class of failure which will be discussed later; this leads from our assumption of component independence. Note that E7 is marked with a small diamond to indicate an undeveloped event. See Figure 9.

Returning to Figure 8, we see that event E4 "Router failed" is a *state-of-component* fault so we expand it into primary, secondary, and command faults all feeding into an or-gate. We find two primary faults E8 "Router hardware failure" and E9 "Router software failure," two secondary faults E10 "Router overloaded" and E11 "Router loses AC power," and one command fault E12 "Router misconfigured."

We mark E8 and E9 as basic events and events E10, E11, and E12 as undeveloped events. The distinction is somewhat arbitrary since we can choose to expand any of these events later. In this case, we decide that we may want to create more detailed

[1]More commonly known as a breadth-first (*vs.* depth-first) expansion.

[2]OOS: Out Of Service

models of power failure, router misconfiguration, and router overload later and that we're satisfied with the rather gross categorizations of router hardware and software failure. Also, the router hardware and software are determined by the router vendor; we ostensibly control AC power, network traffic, and router configuration. Regardless of our categorization of events as basic or undeveloped, we have finished modeling intermediate event E2 as a combination of primary events.

Event E6 "No MTA available" is a *state-of-system* fault. We model E6 as E13 "MTA #1 failed" and E14 "MTA #2 failed" both feeding into an and-gate. Note that if we added a third identical MTA, we could simply duplicate the tree structure beneath E13 under a new event "MTA #3 failed." The common-cause failure event E7 should not change. Caution must be used when copying parts of the tree to ensure that nothing has been missed. The value of the Complete-the-Gate Rule is obvious here.

Following transfer gate 3 from Figure 9 to Figure 10, we can now expand event E13 "MTA #1 unavailable." We can see by our physical diagram and our assumption that the mail servers are identical that the expansion of events E13 and E14 will be similar. E13 is similar to E2, in that E13 is a *state-of-system* fault, composed of a "system failed" and a "system OOS" event combined into an or-gate (events E15 "MTA #1 failed" and E16 "MTA #1 OOS for maintenance.") Similarly, we see event E14 "MTA #2 unavailable" expands into E17 "MTA #2 failed" and E18 "MTA #2 OOS for maintenance," also combined into an or-gate in Figure 11.

Event E15 "MTA #1 failed" is a *state-of-component* fault so we expand it into primary, secondary, and command faults all feeding into an or-gate. We find two primary faults, E19 "MTA #1 hardware failure" and E20 "MTA #1 software failure," two secondary faults, E21 "MTA #1 out of resources" and E22 "MTA #1 loses AC power," and one command fault E23 "MTA #1 misconfigured." This is similar in structure to E4 "Router failed" but we've used "system out of resources" instead of the more specific "system overloaded" to model events where the mail server software does not have the resources to accept all the mail it's receiving (i.e., there's too much incoming mail or there aren't adequate resources available.) At some point, we may wish to specify resources and the effects of their depletion but in this simple model we will not expand these events further.

Similarly, Event E17 "MTA #2 failed" expands into two primary faults, E24 "MTA #2 hardware failure" and E25 "MTA #2 software failure," two secondary faults, E26 "MTA #2 out of resources" and E27 "MTA #2 loses AC power," and one command fault E28 "MTA #2 misconfigured." Our model is complete now that all the leaf nodes of the tree are primary events (either basic or undeveloped events.) We can now generate the cutsets for this tree.

Discussion of Tree Construction

Events E11, E22, and E27 all involve the loss of AC power and may be considered identical according to our assumption that all devices are on the same electrical supply. In a more detailed analysis, one would probably break out electric power into its own fault tree, analyzing off-site power, backup generators, uninterruptible power supplies (UPS), switchgear, and power distribution units.

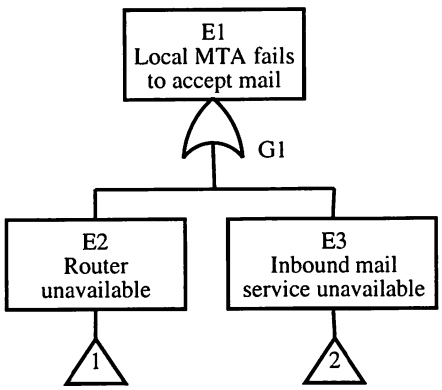


Figure 7: MX1 tree 1 (of 5).

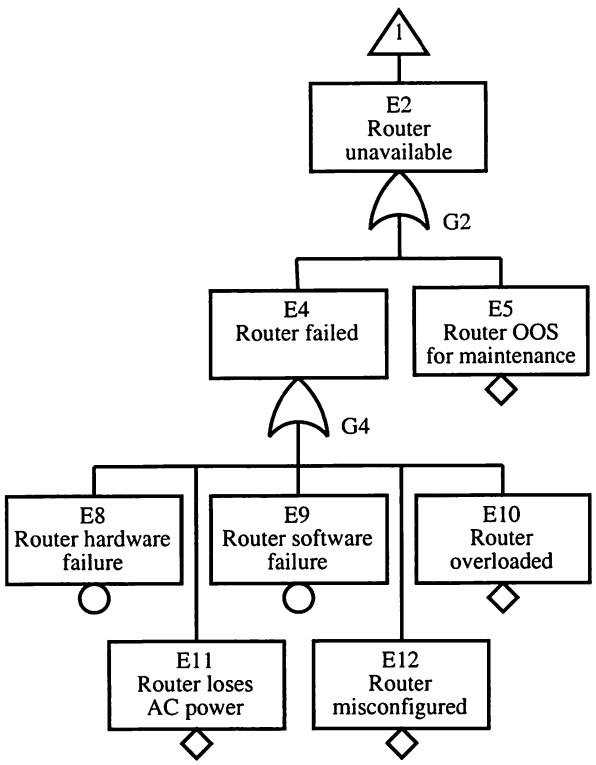


Figure 8: MX1 tree 2 (of 5).

Finding Minimal Cutsets

Having developed a fault tree for MX1, we now can find the minimal cutsets of this tree. In a more complex analysis, one would use a computer code to find the minimal cutsets. In this case we will manually derive them from logical expressions.

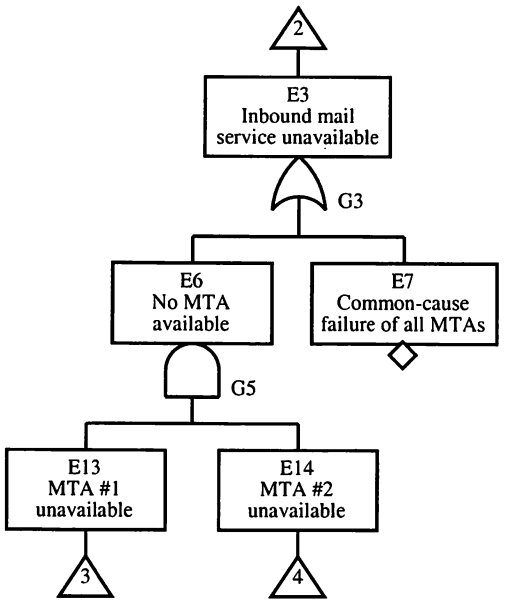


Figure 9: MX1 tree 3 (of 5).

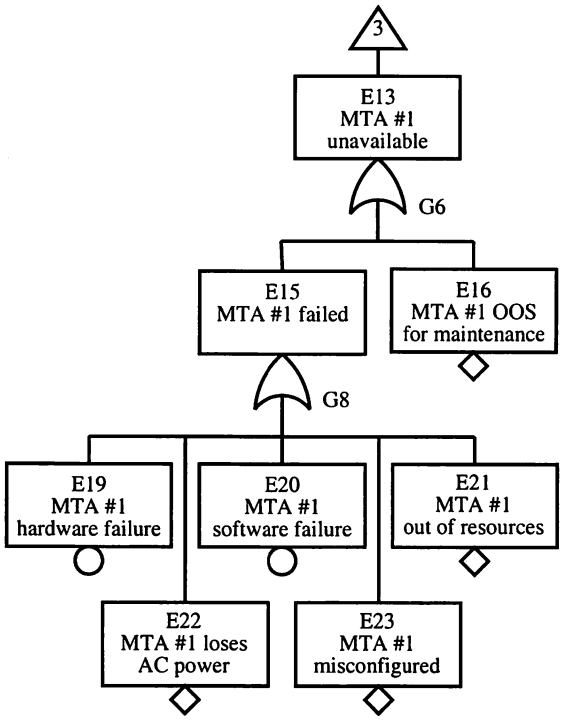


Figure 10: MX1 tree 4 (of 5).

The MX1 fault tree reduces to the following set of equations, where E_i represents the ith intermediate event, e_i represents the ith primary (basic or undeveloped) event and $i$is the event number in the fault tree, e.g., e_5 represents event E5 "Router OOS for maintenance."

$$E_1 = or(E_2, E_3)$$
$$E_2 = or(E_4, e_5)$$
$$E_3 = or(E_6, e_7)$$
$$E_4 = or(e_8, e_9, e_{10}, e_{11}, e_{12})$$
$$E_6 = and(E_{13}, E_{14})$$
$$E_{13} = or(E_{15}, e_{16})$$
$$E_{14} = or(E_{17}, e_{18})$$
$$E_{15} = or(e_{19}, e_{20}, e_{21}, e_{22}, e_{23})$$
$$E_{17} = or(e_{24}, e_{25}, e_{26}, e_{27}, e_{28})$$

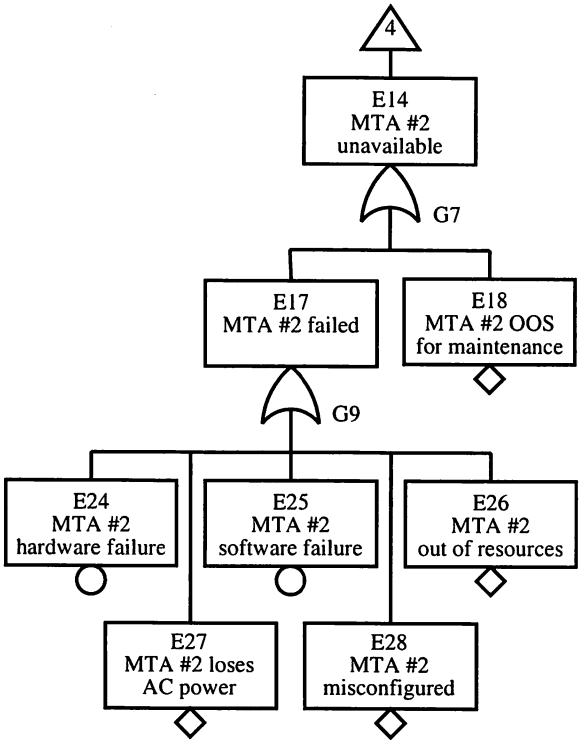


Figure 11: MX1 tree 5 (of 5).

We write top level event E_1 in terms of primary events, using $and(event_1, \ldots, event_i)$ and $or(event_1, \ldots, event_i)$ to represent the logical operations and and or.

$$\begin{aligned} E_1 &= or(E_2, E_3) \\ &= or(or(E_4, e_5), or(E_6, e_7)) \\ &= or(or(e_8, e_9, e_{10}, e_{11}, e_{12}), e_5, and(E_{13}, E_{14}), e_7) \\ &= or(e_5, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}, \\ &\qquad and(or(E_{15}, e_{16}), or(E_{17}, e_{18})) \end{aligned}$$

$$\begin{aligned} E_1 = or(&e_5, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}, \\ &and(or(e_{16}, e_{19}, e_{20}, e_{21}, e_{22}, e_{23}), \\ &or(e_{18}, e_{24}, e_{25}, e_{26}, e_{27}, e_{28})) \end{aligned}$$

Note that $and(or(A, B), or(C, D)) = or(and(A, C), and(A, D), and(B, C), and(B, D))$. In set notation, E_1 in the following cutsets (note: $e_1 e_2 = and(e_1, e_2)$):

$$\begin{aligned} E_1 = or(&e_5, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}, \\ &e_{16}e_{18}, e_{16}e_{24}, e_{16}e_{25}, e_{16}e_{26}, e_{16}e_{27}, e_{16}e_{28} \\ &e_{19}e_{18}, e_{19}e_{24}, e_{19}e_{25}, e_{19}e_{26}, e_{19}e_{27}, e_{19}e_{28} \\ &e_{20}e_{18}, e_{20}e_{24}, e_{20}e_{25}, e_{20}e_{26}, e_{20}e_{27}, e_{20}e_{28} \\ &e_{21}e_{18}, e_{21}e_{24}, e_{21}e_{25}, e_{21}e_{26}, e_{21}e_{27}, e_{21}e_{28} \\ &e_{22}e_{18}, e_{22}e_{24}, e_{22}e_{25}, e_{22}e_{26}, e_{22}e_{27}, e_{22}e_{28} \\ &e_{23}e_{18}, e_{23}e_{24}, e_{23}e_{25}, e_{23}e_{26}, e_{23}e_{27}, e_{23}e_{28}) \end{aligned}$$

Each term in the *or*() clause is a cutset implied by the top event E1. This is neither the set of all cutsets nor the set of minimal cutsets. To generate the set of minimal cutsets we need to remove all cutsets that contain other cutsets. Our assumption about a common power supply allows us to consider events E11, E22, and E27 to be identical. This reduces E_1 to:

$$\begin{aligned} E_1 = or(&e_5, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}, \\ &e_{16}e_{18}, e_{16}e_{24}, e_{16}e_{25}, e_{16}e_{26}, e_{16}e_{28} \\ &e_{19}e_{18}, e_{19}e_{24}, e_{19}e_{25}, e_{19}e_{26}, e_{19}e_{28} \\ &e_{20}e_{18}, e_{20}e_{24}, e_{20}e_{25}, e_{20}e_{26}, e_{20}e_{28} \\ &e_{21}e_{18}, e_{21}e_{24}, e_{21}e_{25}, e_{21}e_{26}, e_{21}e_{28} \\ &e_{23}e_{18}, e_{23}e_{24}, e_{23}e_{25}, e_{23}e_{26}, e_{23}e_{28}) \end{aligned}$$

This is the set of minimal cutsets for tree MX1.

Discussion of Cutsets

We've identified seven single points of failure (SPOF) and 25 two-event cutsets. Six SPOFs relate specifically to the router and one to the electrical power system. The two-event cutsets are combinations of events that fail both mail servers.

While simple inspection of Figure 6 shows the router as a SPOF, our systematic analysis picks out specific faults (hardware, software, configuration) which we might use to drive policy decisions. For example, we may have stricter controls on router configuration than on mail server configuration. We might also require that any mail server configuration changes be applied only to the primary server and only be applied to the secondary server after some period of live testing or 'burn-in' to reduce the risk of common-cause failures due to server misconfiguration. Usually configuration changes are made to increase functionality, security, or reliability so we must weigh the benefits of standardization with the risks of increased common-cause failure. There is no easy answer to this policy question, though fault tree analysis has helped to make this question more apparent. Although we have no quantitative measure of risk at this point in the analysis, we have at least enumerated the risks to the system within the bounds of our analysis.

Note also that we consider unavailability due to planned maintenance activities. The cutset $e_{16}e_{18}$ ("MTA #1 OOS for maintenance," "MTA #2 OOS for maintenance") may be prohibited by procedure; if it isn't, it should be. One must balance the benefits of periodic scheduled maintenance with the risk posed by these activities. Also, it is inappropriate to discard this cutset simply because the combination of events is forbidden by procedure. A proper analysis will consider human error and failure to follow procedure. Errors of omission (not taking the appropriate action) and errors of commission (taking an inappropriate action, or performing the appropriate action at the wrong place or time) and other facets of human reliability analysis (HRA) are beyond the scope of this paper but must be mentioned for completeness. Depending on the thoroughness of the analysis, one may need to evaluate procedures, operational documentation, and user interfaces for potential human reliability pitfalls.

Common-cause failure

While we often assume that primary (component) failures are independent of each other, this is not always the case. Sometimes a system may fail from multiple basic failures attributable to a common root cause. Two examples: 1) components share a common source of electric power, and 2) a set of components produced by a given manufacturer contain an endemic flaw. We generally cannot find common cause failures just by evaluating the fault tree. We must investigate minimal cutsets individually. It is helpful to compare each cutset to a list of common cause categories such as:

- Manufacturer
- Location
- Regional environmental conditions such as susceptibility to flood, seismic activity, tornados, and ice storms
- Local environmental conditions such as temperature, vibration, humidity, dirt, dust, smoke, fire, and R/F interference
- Human interactions (users and operators)
- Degradation due to test or maintenance activities

For each component, we list applicable characteristics in each category, then group components according to similar characteristics. We then identify each minimal cutset susceptible to common cause failures in each group of components and judge if it warrants further analysis. For example, if the router and both mail servers were located in the same cabinet in a data center, they are susceptible to failures stemming from cabinet wiring faults, physical shock or damage to the cabinet, loss of cooling, mistaken identity (i.e., MTA #1 is mistaken for MTA #2, especially if a cryptic machine naming convention is used or the machines are physically similar.) Common-cause analysis is time-consuming and difficult, though [19] provides a more systematic process for identifying common-cause failures.

Obtaining Failure Rate Data

One can estimate failure rates by reviewing operator shift reports, monitoring system logs, using vendor-supplied MTBF estimates, or using engineering judgment. It is important to use measured, installation-specific data whenever practical. Often generic data and engineering judgment are used until one obtains enough information to build a local reliability database.

Note that this sort of data analysis is a fairly involved topic on its own. Probability distribution modeling occupies an entire chapter in the Fault Tree Handbook, and features prominently in [7, 19, 20, and [9].

Estimating Availability

For the purposes of our example, we will make some reasonable assumptions about system operation and use them to estimate component availability. Assume the mail servers operate continuously except for a one hour outage every three months to apply kernel patches, a three hour outage every year to upgrade the mail server software, and 10 minutes of unavailability a month to update configuration files. The probability of a mail server being unavailable due to maintenance is

$$\begin{aligned} t^{MTA}_{maint} &= 4\ [events/yr] \cdot 60\ [min/event] \\ &\quad + 1\ [events/yr] \cdot 180\ [min/event] \\ &\quad + 12\ [events/yr] \cdot 10\ [min/event] \\ &= 540\ [min/year] \\ t^{MTA}_{demanded} &= 60\ [min/hr] \cdot 24\ [hr/day] \cdot 365\ [day/yr] \\ &= 525600\ [min/yr] \\ P_{maint}(MTA) &= \frac{t^{MTA}_{maint}}{t^{MTA}_{demanded}} \\ &= \frac{540}{525600} \\ &= 1.03 \times 10^{-3} \ . \end{aligned}$$

For the router, we assume an annual three hour outage to upgrade firmware and a quarterly five minute outage for configuration changes, giving an "OOS for maintenance" probability of

$$\begin{aligned} t^{router}_{maint} &= 1\ [events/yr] \cdot 180\ [min/event] \\ &\quad + 4\ [events/yr] \cdot 5\ [min/event] \\ &= 200\ [min/yr] \\ t^{router}_{demanded} &= 525600\ [min/yr] \\ P_{maint}(router) &= \frac{t^{router}_{maint}}{t^{router}_{demanded}} \\ &= \frac{200}{525600} \\ &= 3.81 \times 10^{-4} \ . \end{aligned}$$

Performance Metrics and Observed Failure Data

Estimated failure probabilities are useful for scoping studies or design-phase analyses but they are no substitute for observed, site-specific data. Much work has been done in the field of failure rate estimation; references [19, 20] provide a good introduction to common data analysis techniques.

Components rarely have a constant failure rate as shown in the "OOS for maintenance" analysis above. Failure rate generally varies with time. For physical components, the plot of failure rate versus time often takes on a characteristic "bathtub" shape. Three regions of interest are shown in Figure 12 – Region I is known as the "infant-mortality" or "burn-in" region, Region II is a region of nearly constant failure rate, and Region III is the "wear-out" region in which the failure rate increases as components degrade due to wear.

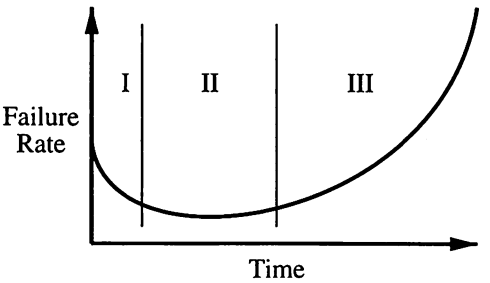


Figure 12: Failure rate variation with time ("Bathtub Curve").

When modeling component failure rates, we must consider the type of component (software, hardware, mechanical, electrical, solid-state), operation characteristics (used continuously or on-demand), maintenance frequency, test frequency and severity,[3] and operating environment. These factors influence the choice of probability distribution used to model the failure rate.

Advanced Models

Calculating the failure rate of repairable systems can be quite complex, especially systems with redundant backups and multiple operation modes. These situations lead to state machine models requiring continuous Markov analysis for solution. It is not uncommon for analysts to conservatively assume systems cannot be repaired just to simplify the analysis.

Quantifying the Model

A Simplified Model

Let us only consider hardware failures, common cause failure, and maintenance activity in our model, e.g., the probability of all other basic events is zero (alternately, we may treat them as external or "house" events and set them to false.)[4] Then our model reduces to

$$E_1 = or(e_5, e_7, e_8, and(e_{16}, e_{18}), and(e_{16}, e_{24}), and(e_{19}, e_{18}), and(e_{19}, e_{24}))$$

where

	Event	Probability
e_5	Router OOS for maintenance	3.81×10^{-4}
e_7	Common cause failure of both MTAs	1.00×10^{-5}
e_8	Router hardware failure	1.00×10^{-4}
e_{16}	MTA1 OOS for maintenance	1.03×10^{-3}
e_{18}	MTA2 OOS for maintenance	1.03×10^{-3}
e_{19}	MTA1 hardware failure	2.0×10^{-2}
e_{24}	MTA2 hardware failure	2.0×10^{-2}

Probabilities of events e_7, e_8, e_{19}, and e_{24} are estimated using engineering judgment. With time, we

[3]This is especially important for backup diesel generators. Rapid start times and frequent testing cause substantial maintenance problems. Ironically, the tests designed to assure availability may tend to reduce it.

[4]There is a distinction between events which don't occur (set to false) and events that occur with zero probability.

would replace probabilities of events e_8, e_{19}, and e_{24} with observed data and perform additional common-cause analysis to estimate the probability of event e_7.

Probability Theory

Quantifying a fault tree is different than quantifying an event tree since the logical combination of probabilities requires some knowledge of basic set theory and Boolean algebra. Readers unfamiliar with these topics are encouraged to review a reference (any of [2, 19, 20, 21, 5, 6]) for more information.

Before converting a logical model into a probabilistic model, we must understand how to model individual logic gates in terms of probability.

The or-*gate*

The or-gate represents the union of the events attached to the gate. If events A and B are inputs of an or-gate and event Q is the output, that is, $Q = or(A, B)$, then the probability of event Q (i.e., $Pr(Q)$) is given by

$$\begin{aligned} Pr(Q) &= Pr(A) + Pr(B) - Pr(A \cap B) \\ &= Pr(A) + Pr(B) - Pr(A)Pr(B|A) \\ &= Pr(A) + Pr(B) - Pr(B)Pr(A|B) \end{aligned}$$

where $Pr(B|A)$ is the conditional probability of B occurring, given that A has occurred.

Some important results from probability and set theory:

- If A and B are mutually exclusive events then $Pr(A \cap B) = 0$ and $Pr(Q) = Pr(A) + Pr(B)$
- If A and B are independent events then $Pr(B|A) = Pr(B)$ and $Pr(Q) = Pr(A) + Pr(B) - Pr(A)Pr(B)$
- If event B is completely dependent on event A then $Pr(B|A) = 1$ and $Pr(Q) = Pr(B)$
- In all cases, one may conservatively estimate $Pr(Q) \cong Pr(A) + Pr(B) \geq Pr(A) + Pr(B) - Pr(A \cap B)$. That is, any error introduced by neglecting $Pr(A \cap B)$ increases $Pr(Q)$ and is therefore conservative.
- For small values of $Pr(A)$ and $Pr(B)$, say < 0.1, $Pr(A \cap B)$ is small compared to $Pr(A) + Pr(B)$ so there is little error in estimating $Pr(Q) \cong Pr(A) + Pr(B)$, provided A and B are independent. This is known as the *rare event approximation.*

The exclusive-or-*gate*

If events A and B are inputs into an exclusive-or-gate and event Q is the output, that is, $Q = xor(A, B)$, then $Pr(Q)$ is given by

$$Pr(Q) = Pr(A) + Pr(B) - 2Pr(A \cap B)$$

If we compare the numerical probability results for the or-gate with those for the exclusive-or-gate, we see that the difference is negligible if A and B are independent. In all cases, treating exclusive-or-gates as standard (inclusive) or-gates is conservative. For this reason, exclusive-or-gates are rarely seen in fault trees.

The and-*gate*

The and-gate represents the intersection of the events attached to the gate. If events A and B are inputs into an and-gate and event Q is the output, that is, $Q = and(A, B)$, then $Pr(Q)$ is given by

$$\begin{aligned} Pr(Q) &= Pr(A)Pr(B|A) \\ &= Pr(B)Pr(A|B) \end{aligned}$$

where $Pr(B|A)$ is the conditional probability of B occurring, given that A occurred. From probability theory, we find:

- If A and B are independent events then $Pr(B|A) = Pr(B)$, $Pr(A|B) = Pr(A)$ and $Pr(Q) = Pr(A)Pr(B)$
- If A and B are not independent, $P(Q)$ may be much greater than $Pr(A)Pr(B)$, though no greater than the larger of $Pr(A)$ or $Pr(B)$.
- If B is completely dependent on event A then $Pr(B|A) = 1$ and $Pr(Q) = Pr(A)$

Quantification

Combining the model and the basic event probabilities, the failure probability of the system is

$$\begin{aligned} E_1 &= or(e_5, e_7, e_8, and(e_{16}, e_{18}), and(e_{16}, e_{24}), \\ &\quad and(e_{19}, e_{18}), and(e_{19}, e_{24})) \\ Pr(E_1) &\cong Pr(e_5) + Pr(e_7) + Pr(e_8) \\ &\quad + Pr(e_{16})Pr(e_{18}) + Pr(e_{16})Pr(e_{24}) \\ &\quad + Pr(e_{19})Pr(e_{18}) + Pr(e_{19})Pr(e_{24}) \\ &\cong 3.81 \times 10^{-4} + 1.00 \times 10^{-5} + 1.00 \times 10^{-4} \\ &\quad + (1.03 \times 10^{-3} \cdot 1.03 \times 10^{-3}) \\ &\quad + (1.03 \times 10^{-3} \cdot 2.00 \times 10^{-2}) \\ &\quad + (2.00 \times 10^{-2} \cdot 1.03 \times 10^{-3}) \\ &\quad + (2.00 \times 10^{-2} \cdot 2.00 \times 10^{-2}) \\ &\cong 9.33 \times 10^{-4} \end{aligned}$$

This result assumes all events are independent and uses the failure probabilities listed at the beginning of this section. A number of simplifying assumptions are made, chiefly the rare event approximation. It is left for the reader to derive the full analytical expression for $Pr(E_1)$ and to compare the true numerical value to the result approximated here (note that the full expression for $Pr(E_1)$ has over 120 terms.) Here we see the value of using a computer code to evaluate and quantify fault trees. While this technique may be performed manually, the calculations for even a simple model quickly become tedious.

A final note on quantification: when combining fault trees and event trees, be sure to combine cutsets and eliminate non-minimal cutsets for each event tree end state before quantifying. Fault trees should be as independent as possible but need not be completely independent, provided that redundant and impossible cutsets are removed before generating numerical results.

Importance Ranking

Now that we have determined the minimal cutsets and have quantified the tree, we can quantitatively assess the importance of each component.

Popular importance measures are Birnbaum and Fussell-Vessely (named after their inventors), risk

achievement worth (RAW) and risk reduction worth (RRW.) Birnbaum importance measures the sensitivity of risk to changes in component reliability over the entire range of reliability – from "component always failed" to "component always available." RAW is the fractional increase in risk assuming a particular event always occurs and RRW is the fractional decrease in risk assuming a particular event never occurs.

Using the notation above, Birnbaum importance of event e_j to top event T is given by

$$I_B^j(T) = Pr(T, Pr(e_j) = 1) - Pr(T, Pr(e_j) = 0)$$

We calculate the Birnbaum importance for router hardware failure e_8 as

$$\begin{aligned} I_B^8 &= Pr(E_1, Pr(e_8) = 1) - Pr(E_1, Pr(e_8) = 0) \\ &= 1.000923 - 9.23 \times 10^{-4} \\ &= 1 \end{aligned}$$

And for MTA1 maintenance outages, we calculate Birnbaum importance as

$$\begin{aligned} I_B^{16} &= Pr(E_1, Pr(e_{16}) = 1) - P(E_1, Pr(e_{16}) = 0) \\ &= (2.19 \times 10^{-2} - 9.12 \times 10^{-4}) \\ &= 2.10 \times 10^{-2} \end{aligned}$$

Qualitatively, Birnbaum importance tells us that risk is much more sensitive to changes in router reliability, less so to changes in mail server reliability. Note that since router reliability is already fairly high, Birnbaum importance tells us that system reliability will fall faster with a decrease in router reliability than with a decrease in mail server reliability.

The Fussell-Vessely importance of a component is the sum of the probability of all event sequences (cutsets) containing that component divided by the total risk, i.e., the fraction of total risk related to this component. For the router

$$\begin{aligned} I_{GV}^{router}(E_1) &= \frac{Pr(e_5) + Pr(e_8)}{Pr(E_1)} \\ &= \frac{3.81 \times 10^{-4} + 1.00 \times 10^{-4}}{9.33 \times 10^{-4}} \\ &= 0.516 \end{aligned}$$

and for MTA1

$$\begin{aligned} I_{GV}^{MTA1}(E_1) &= \frac{Pr(e_7) + Pr(e_{16}e_{18}) + Pr(e_{16}e_{24})}{Pr(E_1)} \\ &\quad + \frac{Pr(e_{19}e_{18}) + Pr(e_{19}e_{24})}{Pr(E_1)} \\ &= \frac{1.00 \times 10^{-5} + 1.06 \times 10^{-6}}{9.33 \times 10^{-4}} \\ &\quad + \frac{2.06 \times 10^{-5} + 2.06 \times 10^{-5}}{9.33 \times 10^{-4}} \\ &\quad + \frac{4.00 \times 10^{-4}}{9.33 \times 10^{-4}} \\ &= 0.485 \end{aligned}$$

In this case Fussell-Vessely importance shows router failure contributes slightly more to overall risk than MTA1 failure. The router's high reliability makes up for the lack of a redundant backup router.

Risk achievement worth and risk reduction worth are calculated for each basic event and a component's importance is taken as the maximum of the RAW or RRW for the basic events associated with a component.

$$RAW_i = \frac{Pr(T, Pr(e_i) = 1)}{Pr(T)}$$

$$RRW_i = \frac{Pr(T)}{Pr(T, Pr(e_i) = 0)}$$

These metrics are related to Birnbaum and Fussell-Vessely importance by

$$I_B^j = \left(RAW_j - \frac{1}{RRW_j}\right) Pr(T)$$

$$I_{FV}^j = \left(1 - \frac{1}{RRW_j}\right)$$

Based on the values above, we find

$$\begin{aligned} RAW_5 &= 1072.10 \\ RAW_8 &= 1072.41 \\ RAW_{16} &= 23.51 \\ RAW_{19} &= 23.08 \end{aligned}$$

and

$$\begin{aligned} RRW_5 &= 1.69 \\ RRW_8 &= 1.12 \\ RRW_{16} &= 1.02 \\ RRW_{19} &= 1.82 \end{aligned}$$

Qualitatively, the RAW results show that a decrease in router reliability will affect the system much more than a proportional decrease in MTA1 reliability. The RRW results show that increasing MTA1 hardware reliability is slightly more effective at reducing risk as a proportional reduction in router unavailability due to maintenance. Both RAW and RRW metrics add additional meaning to component reliability trends.

Compare these metrics to an ad hoc analysis which could either claim that the router is most important because it's a single point of failure (SPOF) or that mail servers are most important because of their much higher failure rate. Note that the ad hoc analysis breaks down rapidly as the complexity of the system increases. In this simple case it's not so apparent but if we made the system more complex by adding more mail servers and a standby router, the ad hoc analysis becomes less useful, approaching mere speculation. This is especially true when all SPOFs are found and eliminated.

Regardless of the figure-of-merit used, one must understand what it represents and calculate it consistently. When using FTA as a risk communication tool, it helps to use simple, intuitive importance measures.

Expanding The Fault Tree Model

This model is trivial though not totally contrived. One benefit of explicitly stating our assumptions at the outset is that we can revisit them individually and systematically to expand our model. Some issues not addressed in this simple analysis include:

- human error including errors of omission and errors of commission
- component dependencies (e.g., if MTA1 fails, will the increased load on MTA2 increase MTA2's failure probability? If we patch machines less often to reduce maintenance unavailability, will the probability of software failure increase?)
- recovery actions
- sensitivity and uncertainty analysis, showing how the model reacts to statistical variations and uncertainty in component reliability.

Of these, human reliability is the most important since human error often dominates risk in high-reliability systems. However, the field of human reliability analysis (HRA) is extremely complex, far beyond the scope of this paper.

Comparison between ETA and FTA

Event tree analysis appears simplistic, even obvious. The technique is important for this very reason – event trees clearly communicate failure modes (consequences) and the event sequences that lead to them. ETA provides a straightforward, consistent, systematic approach to modeling complex systems at a high level. This high-level approach provides a basis for more detailed modeling with fault trees.

Fault trees do not model degraded performance well; since they are built on Boolean operations such and and or, fault trees are best at yielding binary results (e.g., success/failure.) Event trees aren't limited to binary outcomes and can show a variety of consequences.

Event trees serve another purpose; they are often used to break up a large analysis into smaller, more manageable parts, simplifying construction and review. And unlike fault trees, event trees clearly show consequences; fault trees are more useful for showing the existence and probability of failure sequences. Fault trees are usually far more detailed than event trees, modeling low-level component failure and human action. This allows components to be ranked according to their contribution to overall risk. Finally, undeveloped events in fault trees explicitly show the limits of the analysis. Both ETA and FTA have their strengths and weaknesses but the combination of the two provides balance and results in a powerful analytical technique.

Limitations of Probabilistic Risk Analysis

PRA requires skilled analysts, a thorough understanding of the systems to modeled, observed or estimated reliability data, and fault tree analysis software. For many systems, the cost of analysis is excessive. One is cautioned against putting too much faith in absolute failure probabilities; the quantitative measures are most effectively used as relative measures of risk (i.e., is component X more important than component Y, or which sequence of events is most likely?)

Both the aerospace and nuclear industries both have a very strong configuration management (CM) culture while formal configuration management is almost nonexistent in the computer industry. This presents us with a serious though not insurmountable challenge – how can we have faith in a model when the system the model based on is so easily changeable?

We can categorize system changes as topological changes and status changes. Topological changes are changes in which components are added, deleted, or rearranged; for example, adding a new web server or moving machines to a new switch or network. These changes require modifying the structure of a fault tree. Status changes are changes to the failure probabilities based on the observed or postulated condition of the system – these changes only modify event probabilities, they do not affect the model's structure.

Since computer systems can produce copious amounts of diagnostic data, I believe one could compensate for status changes by substituting monitoring for strict CM. Also, by decomposing fault trees into modules representing generic components and by mapping network topology [4] and services, we may be able to compensate for topological changes as well.

Note that monitoring, CM, and PRA are complementary approaches. CM is costly and never perfect; monitoring can show where CM is failing and PRA can show where strict CM is warranted (or wasted.)

Finally, PRA does not handle time-dependent failure well. Skilled reviewers are required to assure fault tree completeness. Also, estimating the reliability of redundant, repairable components can become quite complex, leading to state machines that require sophisticated mathematics[5] for solution.

Suggestions for Future Work

Analysis software

Most PRA software is archaic or proprietary and based on my cursory searches, I have found no modern PRA code that runs under anything but DOS or some form of Microsoft Windows [8]. To reach a wider audience of system administrators, we need a freely-available fault tree analysis code, preferably released under a license that allows the source code to be reviewed, modified, and redistributed. The code should support standard databases (MySql, Oracle,

[5]Laplace transforms, discrete and continuous Markov analysis, and other techniques usually forgotten immediately after passing ones final exams, based on the author's experience.

Postgres, etc.) and produce reports and images in common, portable formats.

Generic Model and Failure Rate Data Repository

Unlike the chemical, nuclear, or aerospace industries, the computer industry relies almost completely on commodity components. A repository of generic fault models and failure rate data would simplify fault tree construction and quantification. While generic models and data are no substitute for a careful site-specific analysis, they would speed learning and would help analysts build the framework for a site-specific analysis.

Integrate Monitoring and Management Tools with FTA Software

Provided PRA analysis software and generic models and data were readily available, the next step would be to link common monitoring and management software with a site-specific failure rate database. One could use network analysis tools to mechanically generate system models or confirm the accuracy of existing models. Integration with monitoring and management tools would allow near-realtime risk profiling of systems, similar to refueling outage risk management software used today in the U. S. nuclear industry.

Research Topics

While software fault tree analysis (SFTA) is an active field of research, I know of little system administration research involving PRA.

Security Analysis and Threat Assessment

One can treat security as a subset of reliability. Since PRA was originally developed to tackle the problem of analyzing low-probability high-consequence events, it seems natural to use the technique for security analysis. Bruce Schneier uses fault tree analysis[6] in Chapter 21 of "Secrets and Lies" [3], though he stops short of using fault trees as a probabilistic model. This may be due a lack of publicly-accessible statistics on attacks [7] but I suspect he uses fault trees as a source of data for game-theoretical security models or other cost-risk-benefit analyses, not as probabilistic models.

Some researchers use fault trees to generate the requirements specification for an intrusion detection system [18]. By analyzing protocols and typical implementations, they found potential vulnerabilities and developed their software specification accordingly.

Analyze Common Internet Protocols

Security considerations are often neglected when a protocol is first designed with the intent of adding security features after the protocol has gained public acceptance. Many protocols are easily abusable or may unnecessarily reveal sensitive information to third parties. PRA can be used to analyze common protocols for potential security, privacy, and abuse vulnerabilities.

[6]He calls his fault trees "attack trees." His notation is slightly different than mine but his methodology is similar.

Investigate Common Operating System Process Management Schemes to Model Process Failure

PRA can be used to analyze common process management schemes to suggest ways of increasing the robustness and reliability of existing operating systems. Once could model permissions, resource requirements, use, and depletion, etc. to estimate and improve process reliability. It may also suggest coding standards to help increase software reliability within particular operating environments.

Develop a Common Risk Assessment Notation or Language

One could add risk assessment notation and techniques to UML (Unified Modeling Language.) New software would be more secure and more reliable if risk assessment were considered part of the design process. UML is often used to describe complex relationships within and among systems leading to its popularity as a design and communication tool. It seems natural to extend UML with risk assessment and management notation.

Beyond ETA/FTA

Cause-consequence analysis and decision table analysis may be even better methods for analyzing computer system reliability than ETA/FTA. Tutorials on CCA and decision tables are less accessible than those for ETA/FTA.

Conclusion

PRA is a powerful technique for analyzing and communicating the reliability of complex systems. Yielding both qualitative and quantitative results, PRA provides a rational basis for decisions and resource allocation in the face of complexity and uncertainty.

Acknowledgments

I would like to thank Jim Robinson and Kristin Epley and everyone in Excite@Home's Product Operations staff for giving me the time and motivation to work on this paper. Special thanks go to Loys Bedell, William Salyer, Vicki Bier, and Alexei Novikov for technical review and support. I am forever in the debt of Marrit Ingman and Susan Pinsonneault for editorial review. Finally, I'd like to thank Mark Burgess and William Annis for their constant motivation and encouragement.

Author Information

Bob Apthorpe is a Senior System Administrator in the Product Operations group of Excite@Home, Inc. He is primarily responsible for content service design, support, and reliability but over the past five years has worked on host and network security, traffic analysis, monitoring, automation, forensic operations, incident response, abuse handling, project management, and documentation.

Prior to joining Excite, he worked as a nuclear safety analyst with Entergy Operations at River Bend

Station in St. Francisville, Louisiana. He earned an M.S. and a B.S. in Nuclear Engineering and Engineering Physics from the University of Wisconsin. In his copious free time, he performs with the We Could Be Heroes improvisational comedy troupe in Austin, Texas and practices just a little aikido. He may be reached at arclight@jump.net or apthorpe@excitecorp.com.

References

[1] Leveson, Nancy, "High-Pressure Steam Engines and Computer Software," Presented as a keynote address at the International Conference Software Engineering in Melbourne Australia, http://www.safeware-eng.com/pubs/HiPreStEn.pdf, 1992.

[2] US Nuclear Regulatory Commission, "Fault Tree Handbook NUREG-0492," http://www.nrc.gov/NRC/NUREGS/SR0492/index.html, 1981.

[3] Schneier, Bruce, *Secrets and Lies: Digital Security in a Networked World*, 2000.

[4] Cheswick, Bill, Hal Burch, and Steve Branigan, "Mapping and Visualizing the Internet," *Proceedings of the USENIX Annual 2000 Technical Conference*, June 18-23, 2000.

[5] United States Naval Institute, *Naval Operations Analysis*, p. 249, 1968.

[6] Gonick, Larry, and Woollcott Smith, *The Cartoon Guide to Statistics*, p. 42, 1993.

[7] Moore, David, Geoffrey Voelker, and Stefan Savage. "Inferring Internet Denial-of-Service Activity," To appear in *Proceedings of the 2001 USENIX Security Symposium*, http://www.caida.org/outreach/papers/backscatter/, 2001.

[8] Idaho National Engineering and Environmental Laboratory, "SAPHIRE – Systems Analysis Programs for Hands-on Integrated Reliability Evaluations," http://saphire.inel.gov/, May 23, 2001.

[9] Burgess, Mark, Hårek Haugerud, Sigmund Straumsnes, "Measuring System Normality I: Scales and Characteristics," http://www.iu.hio.no/mark/SystemAdmin/papers/Normal1.pdf, January, 2001.

[10] Burgess, Mark, *On the Theory of System Administration*, http://www.iu.hio.no/mark/SystemAdmin/papers/SysAdmTheory.pdf, March, 2000.

[11] Leveson, N., L. Alfaro, C. Alvarado, M. Brown, E. B. Hunt, M. Jaffe, S. Joslyn, D. Pinnel, J. Reese, J. Samarziya, S. Sandys, A. Shaw, Z. Zabinsky. "Demonstration of a Safety Analysis on a Complex System," Presented at the Software Engineering Laboratory Workshop," NASA Goddard, http://www.safeware-eng.com/pubs/DemSafAn.pdf, December, 1997.

[12] Neumann, Peter G., *Computer-Related Risks*, 1995.

[13] Baker, D. N., J. H. Allen, S. G. Kanekal, and G. D. Reeves, "Pager Satellite Failure May Have Been Related to Disturbed Space Environment," http://www.agu.org/sci_soc/articles/eisbaker.html.

[14] Stevens, W. Richard, *TCP/IP Illustrated, Volume 1: The Protocols*, 1994.

[15] Partridge, Craig, *RFC 974: Mail Routing and the Domain System*, January, 1986.

[16] Postel, Jonathan B., *RFC 821: Simple Mail Transfer Protocol*, August, 1992.

[17] Hazel, Philip, *Exim: The Mail Transfer Agent*, June, 2001.

[18] Helmer, Guy, Johnny Wong, Mark Slagell, Vasant Honavar, Les Miller, and Robyn Lutz, "A Software Fault Tree Approach to Requirements Analysis of an Intrusion Detection System," *Proceedings, Symposium on Requirements Engineering for Information Security*, Indianapolis, IN, http://latte.cs.iastate.edu/ghelmer/SFTA-ID.ps, March, 2001.

[19] Henley, Ernest J., Hiromitsu Kunamoto, *Probabilistic Risk Assessment: Reliability Engineering, Design and Analysis*, 1992.

[20] Billinton, Roy, Ronald N. Allan. *Reliability Evaluation of Engineering Systems: Concepts and Techniques, Second Edition*, 1992.

[21] Grimaldi, Ralph P., *Discrete and Combinatorial Mathematics: An Applied Introduction*, June 1989.

Scheduling Partially Ordered Events In A Randomized Framework – Empirical Results And Implications For Automatic Configuration Management

Frode Eika Sandnes – Oslo University College

ABSTRACT

Automatic configuration management involves maintaining a set of shared and distributed resources in such a way that they serve a community of users fairly, promptly and reliably. In this context, this paper discusses experiments that measure the effect of adding randomized scheduling of partially ordered events to configuration management tools. Three characteristics of randomized scheduling are investigated: efficiency, robustness and security. A configuration management process is efficient if it minimizes the use of resources. It is robust if it is not vulnerable to malicious acts or inadvertent human errors. It is secure if its management model is hidden from observers. Several experiments suggest that randomized scheduling of partially ordered events has advantages over commonly used deterministic strategies, on average producing more efficient schedules. Further, randomized scheduling greatly degrades the accuracy of observer predictions of future behavior. In addition, randomized scheduling obscures the management model such that an observer will have to make a large number of observations in order to obtain the complete management model. The results of the study support the use of randomization in automatic configuration management tools.

Introduction

Several protocols have been designed with distributed system administration in mind. For instance the Simple Network Management Protocol (SNMP) [14, 24], as well as higher level, abstract languages for policy based management [2, 4, 9, 11]. These languages allow the administrator to define what actions to be taken in certain situations. There are also a several tools providing automatic and distributed configuration management such as cfengine [4] or IBM's Tivoli [3, 10, 18, 20]. Typical actions include the creation, copying, modification and deletion of files, setting ownership and permissions and process control.

These tools in some sense understand the concepts of events and responses. An event can for example be a particular time of day or the absence or presence of some entity such as a file or a process. Usually, an event triggers a set of responses from the configuration management tool. Such responses are themselves events and are usually related by a partial ordering that can be represented by a directed acyclic graph. These responses must be performed in a sequence satisfying the partial ordering, via a process called *scheduling*. Current tools such as cfengine version 1 employ very simple scheduling algorithms where events are scheduled in a deterministic fixed order. However, it has recently been suggested that the use of randomization can greatly improve the efficiency and security of a configuration management system [7]. This work supports some of the claims in [7] through a set of three experiments.

The first set of experiments measure the effect of randomized scheduling strategy upon the efficiency of the configuration management process. It is an established fact in the parallel computing literature that scheduling of tasks greatly affects the resource utilization in a distributed system. However, to the best of our knowledge, this has not previously been studied from a completely randomized viewpoint. Most work on scheduling focusses upon specialized scheduling algorithms designed to find optimal schedules, exhibiting maximum efficiency. These algorithms solve *single objective* optimization problems. However, our problem of configuration management can be viewed as a *multi-objective* optimization problem, where the objective is to achieve good efficiency, but also sufficient security, fairness and availability of service.

The second set of experiments measure the extent to which randomization can contribute to the security of the configuration process. As discussed in [7], the configuration management process can be viewed as a competition between forces: destructive forces that attempt to disorder the system, and constructive forces that try to re-order the system. In such systems, the destructive forces may want to predict the next move triggered by the constructive force in order to sabotage the system.

It is trivial for an observer to predict the order of configuration steps if the observer knows the configuration management model a priori. By replacing a

deterministic model with one based on randomness it becomes more difficult for an observer to make accurate predictions. A second set of experiments were set up to quantify, in terms of probabilities, how difficult it is for an observer to predict the configuration management actions under a randomized regime. Obscurity is studied as part of a dynamic and reactive security policy for maintaining resource availability in the presence of resource abuse.

The third set of experiments was designed to quantify the extent to which an observer may monitor configuration management actions and use these observations to reconstruct the management model. The impact of randomization on hiding the model from the observer is studied. In real life, an observer usually does not know the management model beforehand. But the observer may *learn* the management model by continuously observing the system over time and identifying trends. A deterministic fixed-order model is trivial to capture – but what about a random model? The algorithm used in this experiment falls into the same category as algorithms described in a totally independent study by Couch and Daniels [8] where the precedence hierarchy of troubleshooting procedures are uncovered over time through a series of observations.

The three experiments are presented sequentially in self-contained sections with background material, a description of the experimental method, results and discussion. Finally, a discussion on how the results can be used to improve the design of automatic configuration management tools is provided.

However, before examining the details of the experiments, scheduling is discussed in the context of distributed configuration management.

Management, Resource Allocation, and Scheduling

Scheduling takes many forms, such as job-shop scheduling, production scheduling, silicon chip design, multiprocessor scheduling and so on. It can take place within any extent of time, space or other dimension. Scheduling algorithms are usually *dynamic* or *static*.

Dynamic scheduling involves continuously allocating a set of resources to a time-variant problem. Modern operating system kernels provide good examples of dynamic scheduling in the way processes are scheduled and the processor resource is time-sliced. The set of resources – total processing power, primary and secondary storage units, network bandwidth, etc. – remain fixed. The scheduling problem is time-variant as processes are continuously started, stopped, resumed and killed by the users of the system, either explicitly or implicitly. The scheduling objective is to maximize the use of available resources in any given situation.

Static scheduling involves assigning a set of fixed resources to a fixed and constant problem. For example, scheduling timetables in schools and universities involves static scheduling, as different classes of students have to be assigned classrooms and lecture theaters, and students and teachers must not have overlapping timeslots. The classes, the courses and the lecture rooms are known a priori. In general, static scheduling problems are NP hard. Static scheduling involves assigning the vertices (tasks) of an acyclic, directed graph onto a set of resources, such that the total time to process all the tasks are minimized. The actual time it takes to process all the tasks is usually referred to as the *makespan*. An additional objective is often to achieve a short makespan while minimizing the use of resources.

Such multi-objective optimization problems involve complex trade-offs and compromises, and good scheduling strategies are based on a detailed and deep understanding of the specific problem domains. Most approaches belong to the family of priority-list scheduling algorithms, differentiated by the way in which task priorities are assigned to the set of resources. Traditionally, heuristics have been employed in the search for high-quality solutions [13]. Over the last decade heuristics have been combined with modern search techniques such as simulated annealing and genetic algorithms [1].

Scheduling Objectives and Configuration Management

The scheduling problem occurs naturally in distributed configuration management. Within a single configuration rule there is often a set of classes or triggers that are interrelated by precedence relations. These relations constrain the order in which configuration actions can be applied; these graphs can be described formally.

A set of precedence relations can be represented by a directed graph, $G = (V, E)$, containing a finite, nonempty set of vertices, V, and a finite set of directed edges, E, connecting the vertices. The collection of vertices, $V = \{v_1, v_2, \ldots, v_n\}$, represents the set of n configuration actions to be applied and the directed edges, $E = e_{ij}$, define the precedence relations that exists between these configuration actions (e_{ij} denotes a directed edge from configuration action v_i to v_j).

This graph can be cyclic or acyclic. Cyclic graphs consist of *inter-cycle* and *intra-cycle* edges, where the inter-cycle edges are dependencies within a cycle and intra-cycle edges represent dependencies across cycles. Management models in system administration are typically cyclic and the cycles have to be broken prior to scheduling. However, this discussion is limited to acyclic graphs. Literature addressing cyclic graphs include [6, 15, 17, 21, 23]. The reader is also referred to [19] for information on graph algorithms and [22] for general graph theory.

Configuration management is a mixture of *dynamic* and *static* scheduling. It is dynamic in the sense that it is an ongoing real-time process where

configuration actions are triggered as a result of the environment. It is static in the sense that all configuration actions are known *a priori*. Configuration actions can be added, changed and removed arbitrarily and dynamically. However, this does not violate the static model because such changes would typically be made during a time-interval in which the configuration tool were idle or offline. The hierarchal configuration management model remains static, in the reference frame of each configuration, but may change dynamically between successive frames of configuration. See [7] for an in-depth discussion of dynamic and static scheduling in configuration management.

Few studies have been conducted into randomized scheduling as the scheduling objectives usually are to find the most efficient schedules with the shortest makespan. However, in this work efficiency is just one of several goals, and the main emphasis is on improving security and obscuring the information that can be gained by observers by watching the configuration process as it occurs.

Security and Randomization

All scheduling problems are resolved by traversing the graph using *topological sorting*. In simple terms, a topological sort of a directed graph is a list of the vertices of the graph in an order that preserves the precedences in the graph. If the vertices represent tasks, a topological sort of the tasks is an order in which they can be accomplished while satisfying the precedences between them.

Most topological sorting algorithms are based on the concept of a *freelist*. One starts by filling the freelist with the entry nodes, i.e., nodes with no parents. At any time one can freely select, or schedule, any element in the freelist. Once all the parents of a node have been scheduled, the node can be added to the freelist. Scheduling strategies differ in the way elements are selected from the freelist. Most scheduling algorithms attempt to select freelist elements so that the schedule can be completed in the shortest possible time.

A popular heuristic for achieving a short schedule is the Critical Path/Most Immediate Successor First (CP/MISF) [13]. Tasks are scheduled with respect to their levels in the graph. Whenever there is a tie between tasks (when tasks are on the same level) the tasks with the largest number of successors are given the highest priority. The critical path is defined as the longest path from an entry node to an exit node.

In configuration management, the selection of nodes from the freelist is often viewed as a trivial problem, and the freelist may, for instance, be processed from left to right, then updated, in an iterative manner. If instead one employs a strategy such as the CP/MISF, one can make modifications to a system more efficiently in a shorter time than by trivial strategy.

A system can be prone to attacks when it is managed deterministically. By introducing randomness into the system, it becomes significantly harder to execute repetitive attacks on the system. One can therefore use a random configuration action implementation when selecting elements from the freelist. A randomized scheduling algorithm adhering to the above description is outlined later in this article. In the next section the effect of randomized scheduling on efficiency is investigated quantitatively through a series of experiments.

Part I: Efficiency

Configuration management includes activities such as process monitoring and control and the management of files and file-structures, involving operations such as copying, moving, deleting and modification of files. Operations on files are typically amongst the most time-consuming and resource-demanding since they require mechanical movement. There are two fundamental classes of management operations – *local* and *remote*. Local interactions involve operations on files residing on the disk-drives attached to the local machine. Remote operations include file accesses on remote machines accessible via a computer network.

Common to all computer hardware manufactured during the last decades are disk drives and networking peripherals equipped with the well known Direct Memory Access (DMA) controllers. A DMA controller allows content to be transferred asynchronously between the system memory and a peripheral device – such as a disk drive or a network card – without wasting processor cycles. Thus, in true parallel fashion most computers can, for example, copy huge files and do number crunching simultaneously. In this article *asynchronous* operations refer to those that can be initiated and performed in parallel without intervention from the processor. Synchronous operations refer to operations that cannot be performed in the background, or parallel, resulting in a processor in a busy state until the completion of the operation. The notion of time-sharing and multitasking is not included in this discussion as it represents pseudo-parallelism that does not lead to any efficiency gains.

Further, most remote operations can be executed asynchronously on a remote machine. First the local machine issues some form of *remote procedure call* (RPC). The remote procedure call is sent, received, and its parameters un-marshalled at the remote machine, and the remote procedure initiated by the remote processor. While the remote procedure call is in progress at the remote machine the local processor can either wait for the remote procedure call to complete, or perform some other task simultaneously. In this case, the local machine must at some later point in time check that the remote procedure call completed successfully or obtain this information via an interrupt, callback or a signal.

Thus computer systems embody two forms of true parallelism, parallelism within a machine due to parallel peripheral devices and parallelism achieved by the "delegation" of tasks to independent processing nodes on the same network. In a system that provides these forms of parallelism, the sequence in which events are scheduled affects efficiency. This fact is supported by the vast body of literature on parallel processing and task scheduling.

For example, suppose we one is given a hardware configuration consisting of a central computer with two disk drives *A* and *B*, and a remote computer with a disk drive *C*. Suppose the configuration management tasks to be performed include copying a configuration file from disk *A* to disk *B* on the local computer and the same file to disk *C* on the remote computer. Then, the file is to be modified by replacing an identifier string occurring in the file with the hostname of the machine to which the file has been copied. This problem can be broken down into four logical operations:

1. Copy the file from *A* to *B*.
2. Modify the newly copied file on *B*.
3. Copy the file from *A* to *C*.
4. Modify the newly copied file on *C*.

Clearly, there is a partial ordering on the four tasks. Task 1 must precede task 2 and task 3 must precede task 4, while the pairs of tasks (1, 2) and (3, 4) are independent. Further, assume that there is a processing delay associated with each task equalling one, and that there is a delay of 0.1 associated with setting up a local asynchronous operation and a delay of 0.2 associated with setting up a remote asynchronous operation over the network. This simple example yields six valid processing sequences given in Table 1.

The schedules resulting from these sequences are given in Figure 1. Clearly, the makespans vary from 2.5 to 4.5 – the worst makespan being nearly twice as long as the shortest or the *optimal* makespan. Clearly, the sequences 1342 and 3142 are both optimal and symmetric to each other. In the sequence 1342, Task 1 is initiated asynchronously, followed by the remote initiation of the asynchronous task 3. Upon completion of task 3, task 4 is initiated asynchronously followed by synchronous initiation of task 2. This sequence leads to a good exploitation of the available hardware with the least idle time. On the other hand the sequence 1234 (and 3412) yields a poor result since the second task cannot be started before the first task 1 has completed, although task 1 is asynchronous. The second task 2 is synchronous and the third task cannot be started before the processor is available after processing task 2. Further, the final task 4 depends on the completion of task 3.

Why Is Randomness Better?

Assuming that the scheduling order can affect the makespan of the schedule, how can a random order be better than a fixed order? In the traditional scheduling context there are few differences between a fixed order and a random order, because a fixed order is simply one of the random orders. With a fixed order, however, there is a probability of picking a good, medium or poor schedule. With some luck it is a good schedule, but it could equally well be a poor schedule. In any case, since it is a fixed order, and the process is repeated, the scheduling algorithm is locked into this configuration. Thus, the effect of a poor selection is multiplied over time.

Sequence	makespan
1234	4.5
1324	3.5
1342	2.5
3412	4.5
3124	3.5
3142	2.5

Table 1: Valid processing sequences and their makespans.

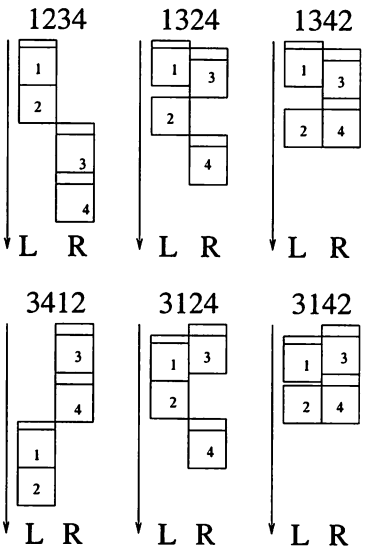


Figure 1: Schedules generated for the six sequences. L denotes Local, R denotes Remote and the vertical axis time.

Conversely, with the random strategy, a different order is selected each time – sampling the entire spectrum of makespans. Thus over all executions the schedules will yield average makespans.

Method

The Graph Test Suite

The suite of random directed acyclic graphs (DAGs) used in the experiments was generated by defining a $n \times n$ square triangular adjacency matrix as follows:

$$A = \begin{bmatrix} 0 & p_{1,2} & p_{1,3} & \cdots & p_{1,n-1} & p_{1,n} \\ 0 & 0 & p_{2,3} & \cdots & p_{2,n-1} & p_{2,n} \\ : & : & : & : & \cdots & : \\ 0 & 0 & 0 & \cdots & 0 & p_{n-1,n} \\ 0 & 0 & 0 & \cdots & 0 & 0 \end{bmatrix} \quad (1)$$

where $p_{i,j}$ is a random integer holding 1 with probability p and 0 with probability $1 - p$ for the element at row i and column j. The triangular nature of the matrix ensures that the graph is directed and acyclic. A non-zero element at row i and column j indicates that event j depends on event i. Vertex i is the parent of vertex j, and vertex j is the child of vertex i.

Two sets totalling 19 graphs were generated, where graph size and density were varied. The size n of the graphs was varied from 10 to 100 vertices in steps of 10, and the graph density was varied in nine steps by adjusting the probability p from 0.1 to 0.9 in steps of 0.1.

A graph with an element probability of 1 has all the upper triangular elements set to 1, and represents a fully connected graph. When all the transitive edges are removed the resulting graph is a linear chain or a completely ordered sequence of events. On the other hand, if the probability is 0 then there are no dependencies between the events and all the events are independent. Thus, there is no ordering. Probabilities in the range of 0.1 to 0.9 yield graphs with varying degrees of partial ordering from the completely ordered linear chain of events to the unordered independent set of events. For an interesting discussion on graph structures see [12].

One may argue that authentic dependence graphs are more realistic than artificial graphs. Authentic graphs fall into one of the classes of commonly occurring graph topologies (trees, meshes etc.). Random graphs allow graph characteristics to be varied enabling crucial relationships to be identified. Such relationships might not be revealed by using a biased set of graphs.

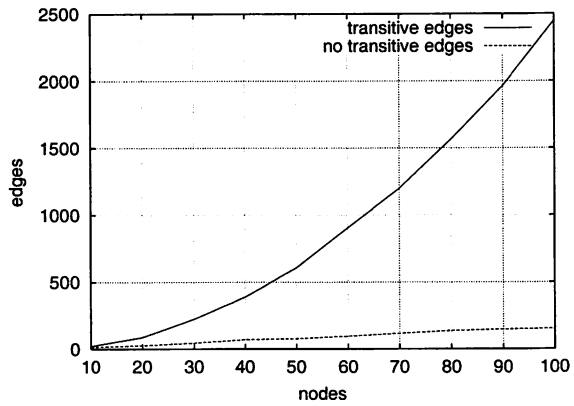


Figure 2: The number of edges in the graph is plotted against the number of vertices in the graph.

Figures 2 and 3 show the number of non-transitive and transitive edges in the suite of random graphs, where the number of edges are plotted against graph size and graph density respectively. Note that the number of transitive edges increases quadratically with graph size while the number of non-transitive edges increases linearly. Also note the interesting fact that although an increase in the adjacency matrix probability leads to an increase in the number of transitive edges, it also leads to a decrease in the number of non transitive edges.

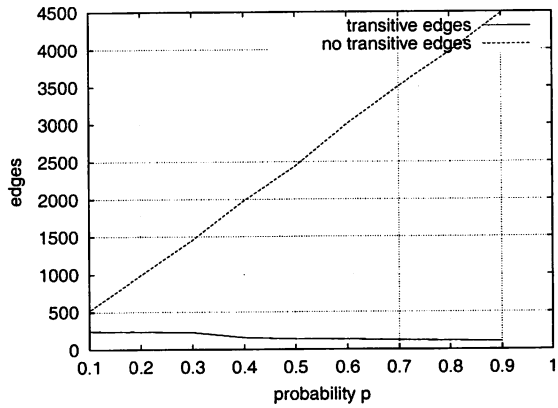


Figure 3: The number of edges in the graph is plotted against the probability p for the elements in the triangular adjacency matrix.

Graph Pre-processing

Random graphs were pre-processed by removing all the *transitive dependencies*. A transitive dependency can be defined as follows; If vertex y depends on x and vertex z depends on y, then z also depends on x. Therefore, a transitive relation such as z depends on x can be removed from the graph since it is *transitively implied* by the two relationships z depends on y and y depends on x.

The purpose of removing the "unnecessary" transitive dependencies is to simplify and purify the graphs. Graphs with no transitive edges are more efficient to process and two graphs are easier to compare if they contain no transitive edges. A graph with transitive edges and a graph with no transitive edges may express the same partial ordering of the vertices, however they have different topological structures. The precedence relations are conserved when the transitive edges are removed.

Transitive edges were removed using the following strategy:

```
For each node in graph
   For each parent of the node
      If parent is in any of the
      ancestors of the other parents
      of the node
         Then remove the edge from
         the node to the parent as
         it is transitive.
```

Figure 4 shows an example of a graph with (left) and without transitive dependencies (right). Clearly, D and E are transitively dependent on A, since D depends on B which again depends on A and E depends on C which also depends on A. Further node F is transitively dependent on both A, B and C, since it depends on D and E.

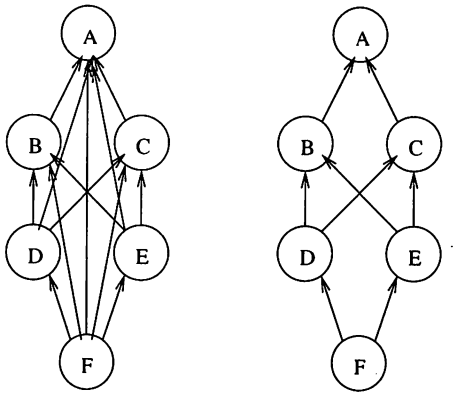


Figure 4: Removing transitive dependencies from a graph.

Scheduling Strategy

In the first experiment the graphs were scheduled using a randomized scheduling algorithm that can be outlined as follows:

```
freelist := all_entry_nodes;
unscheduled := all_nodes;
while (not unscheduled.empty())
  begin
    node := freelist[random];
    process(node);
    scheduled.add(node);
    freelist.remove(node);
    for all nodes in unscheduled whose
    parents are all scheduled
      begin
        freelist.add(nodes);
        unscheduled.remove(nodes);
      end
  end
```

Notice that elements were selected randomly from the freelist. Events were scheduled onto the first available timeslot on the resource or later depending on the completion times of parent tasks, as all parent tasks of a task must have completed before the task can commence. Resource allocations where fixed. Four resources were used and tasks were allocated to a resource with an index matching the modulo 4 of the task index. Each task was given unity execution and communication/setup delays. The resources represented individual DMA controllers managing individual devices such as local disk drives and network cards allowing connectivity to remote disk drives. However, any number of resources greater than one could be employed to demonstrate the differences in efficiency.

Results

The results of the scheduling experiment are shown in Tables 2 and 3. Table 2 lists the number of nodes in the graph, the smallest makespan, the largest makespan, the mean makespan, its variance, and the variation in makespan as a percentage of the longest makespan. Table 3 lists the same data where the first column describes the graph density.

	Position			Spread	
nodes	min	max	mean	var	%var
10	8	8	8.0	0.0	0.0
20	16	19	16.9	1.9	15.7
30	29	38	32.6	6.7	23.6
40	25	36	26.8	8.0	30.5
50	46	55	49.0	5.4	16.3
60	51	63	53.9	8.6	19.0
70	57	65	60.3	4.0	12.3
80	84	91	86.4	5.4	7.6
90	65	85	73.2	21.6	23.5
100	81	98	89.5	10.7	17.3

Table 2: Makespan characteristics with varying graph sizes.

	Position			Spread	
density	min	max	mean	var	% var
0.0108	183	183	183.0	0.0	0.0
0.0116	158	158	158.0	0.0	0.0
0.0122	130	133	131.1	2.1	2.2
0.0134	107	114	109.2	4.0	6.1
0.0139	83	94	88.6	6.3	11.7
0.0159	77	99	84.4	15.6	22.2
0.0228	50	68	59.2	16.1	26.4
0.0237	36	57	44.2	20.2	36.8
0.0244	37	53	43.8	16.6	30.1

Table 3: Makespan characteristics with varying graph densities.

Discussion

Table 2 shows that the makespan generally increases with an increase in the number of nodes. This is logical, as if one is increasing the workload without adding resources it will take longer to service the jobs. Also, the variance in makespans do not change significantly with the graph size. Thus, there is room for improvement, regardless of the graph size.

Table 3 is even more interesting. When the graph size is fixed and the graph density is increased there is a seemingly linear reduction in makespan. This is because a more connected graph in general provides more parallelism than a less connected graph, and this parallelism is thus exploited during scheduling. Further, the variance of the graph increases with the density. Thus, a dense graph is both more efficiently scheduled but also provides more variance in its makespan. The presence of variance in makespan proves that the order in which the tasks are scheduled have an significant effect on the makespan, and, thus the efficiency of carrying out the configuration management task. From this one can conclude that a dense graph provides more room for improvement than a sparse graph. As it also results in shorter makespans, a management topology should be designed with a maximum number of non-transitive dependencies with the view to improve the performance of the configuration management task.

Finally, this experiment confirm that the processing order of the elements has an impact on performance and that the random strategy can produce better results than a fixed-order strategy.

Part II: Malicious Intervention

In this part of the experiment three assumptions are made. First, an observer has complete knowledge of the configuration management structure (precedence relations). Second, an observer is able to monitor transactions. Third, the observer is capable of, and has the desire to, intervene. The scenario can be viewed as a game between two parties; see Burgess [5] for a discussion on game theory applied to configuration management. The system administrator, or operator, is responsible for maintaining the operation of his or her computer system, ensuring a high quality of service to its users or subscribers. The observer possesses the ill-intended desire of sabotaging the operation of the computer system. The operator issues an action and the observer tries to pre-empt the action by a guess or a prediction. If the operator follows a predictable pattern, the observer will know the next move of the operator. However, if the operator employs a non-deterministic pattern then it is difficult for the observer to predict the next move, and the prediction becomes a gamble.

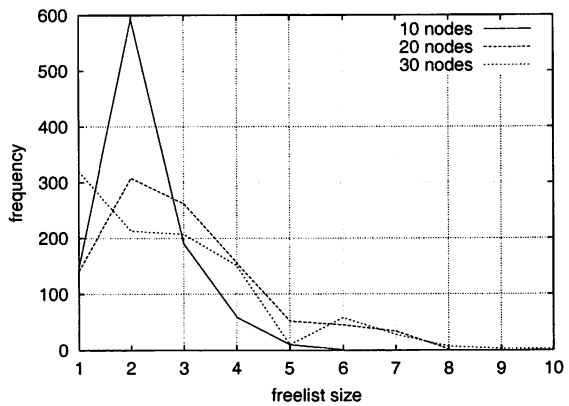


Figure 5: The distribution of freelist sizes over 1000 iterations. Varying graph sizes (10, 20 and 30 vertices).

This can be illustrated with a practical example. Imagine that a computer system consist of two temporary storage areas: /tmp and /audio/tmp, each cleared at regular times. Suppose that the operator has configured a configuration action where the /tmp directory is cleared every night at 2 am, and /audio/tmp is cleared every day at 6 am. Further, there is an inconsiderate user that wants to store huge amounts of data exceeding the allowed quota. A well-known trick is to use a shared temporary areas. Such areas usually have large capacities. The result is that one user hogs the temporary space of other users, preventing them from carrying out their work. Further, if this ill-intended user knows the cleanup configuration action, it is quite easy to sustain this antisocial state by moving the files around to prevent the files from being deleted. The files are moved from /tmp to /audio/tmp sometime in the interval between 6 am and 2 am the following night, and in the interval between 2 am to 6 am the files are moved back from /audio/tmp to /tmp. Thus accurate predictions of the management activities can be exploited by the observer. The result is that the user gets away with unfair exploitation of resources. However, if a random strategy is applied to scheduling these two maintenance events, then it becomes impossible for the observer to predict the next move accurately. The observer is not able to know whether, or when, the /tmp or /audio/tmp directory are cleared. The probability of the malicious user losing his or her data in this situation is 0.5, and at the next iteration the probability is 0.5 and so forth. And the total probability of keeping the data is therefore 0.5^i where i are the number of iterations. In general,

$$\lim_{i \to \infty} p^i = 0 \qquad (2)$$

Thus, the random strategy has a measurable reinforcing effect on the sturdiness of the system.

The objective of this experiment was to evaluate what is gained by introducing randomness into the scheduling of the configuration management actions and how much randomness that will typically be available.

If an operator is given a choice of k actions and one is chosen randomly, then the observer is able to guess or predict the next move with probability $p = 1/k$. Clearly, it is desirable to operate with a small probability p as possible. Thus the scheduling framework should ideally be designed to maximize k.

Method

For each of the randomly generated graphs in the test suite, 1000 valid random sequences were generated through random scheduling. For each iteration of the scheduling algorithm, the size of the freelist was recorded, where the freelist contains the list of possible alternatives. The data obtained for each graph were used to generate a set of histograms, illustrating freelist size distributions for the different graph configurations.

Results

Figures 5, 6, 7, and 8 depict the distribution of freelist sizes obtained over 1000 iterations. The different plots represents graphs with a varying number of vertices.

Figures 9, 10 and 11 demonstrate the distribution of freelist sizes obtained over 1000 iterations. The different plots represents graphs with a varying graph density.

Tables 4 and 5 summarize the results for the size experiment and the density experiments respectively. The first column describes the experiment parameter (size and density). The second column shows the

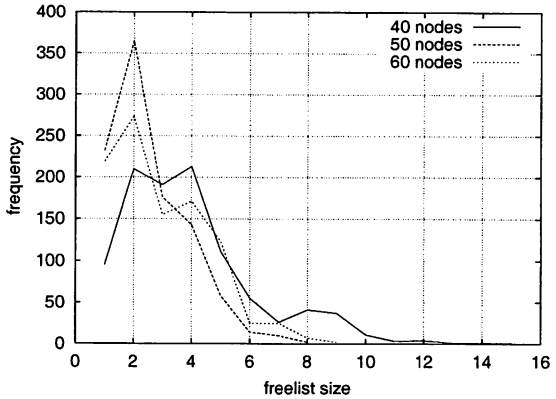


Figure 6: The distribution of freelist sizes over 1000 iterations. Varying graph sizes (40, 50 and 60 vertices).

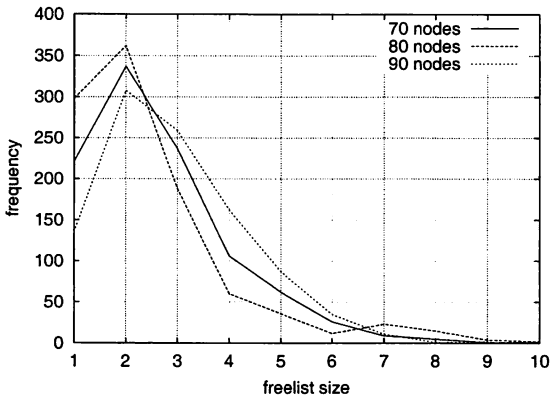


Figure 7: The distribution of freelist sizes over 1000 iterations. Varying graph sizes (70, 80 and 90 vertices).

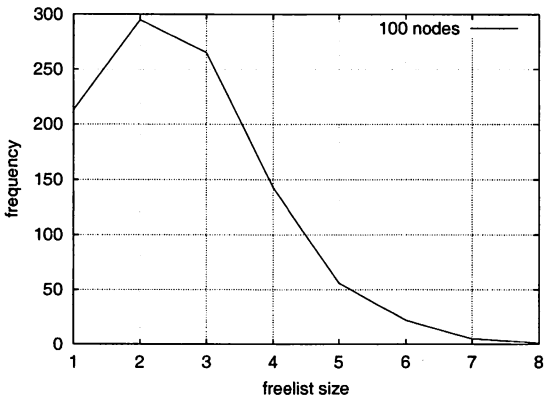


Figure 8: The distribution of freelist sizes over 1000 iterations. Varying graph sizes (100 vertices).

percentage of freelists with size 1 (the percentage of completely deterministic scheduling situations). Column three shows the percentage of freelists with a size greater than one (the percentage of non-deterministic scheduling situations). Column four shows the positions of the peak in the distributions. Column five

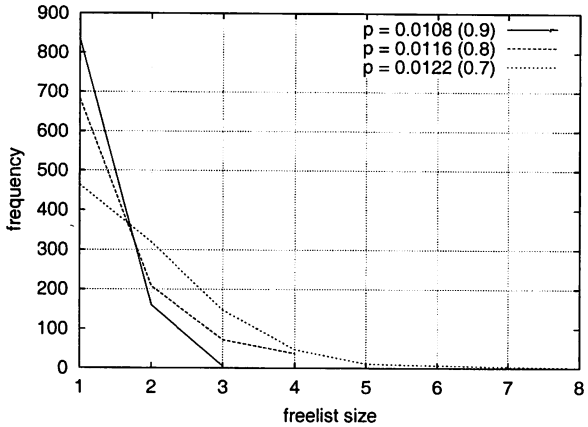


Figure 9: The distribution of freelist sizes over 1000 iterations. Varying graph densities (0.0108, 0.0116 and 0.0122).

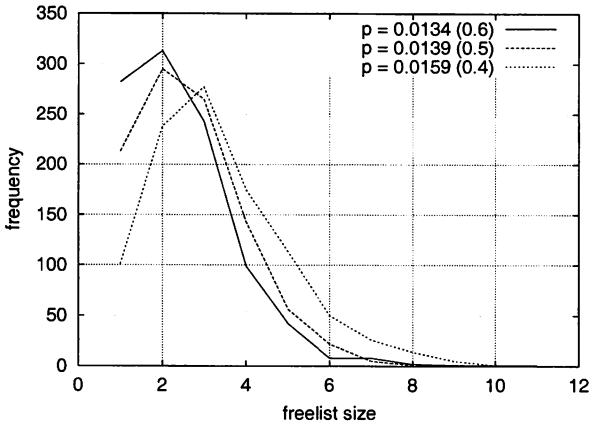


Figure 10: The distribution of freelist sizes over 1000 iterations. Varying graph densities (0.0134, 0.0139 and 0.0159).

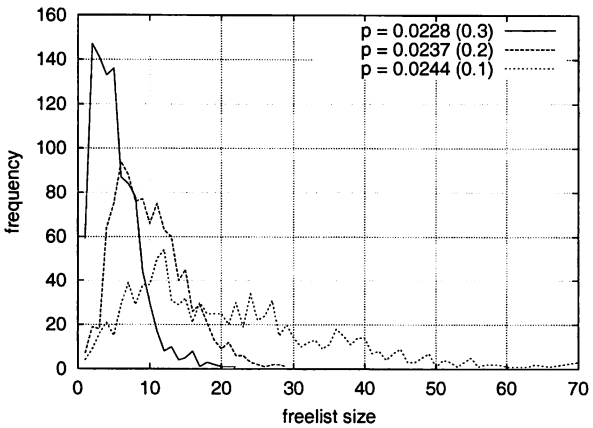


Figure 11: The distribution of freelist sizes over 1000 iterations. Varying graph densities (0.0228, 0.0237 and 0.0244).

shows the magnitudes of the distribution peaks and the final column shows the widths of the distributions.

Discussion

The results indicate that the distributions of freelist sizes for smaller graphs are narrow with most freelist vectors of size 2 (59.3%) and some freelists with size 1 and 3 elements (about 20% each). As the number of nodes in the graphs is increased the distribution is smeared outwards spanning a larger sized freelist. The distribution peaks are still at 2, but the peaks are smaller at approximately 30%. For example, the graph with 100 nodes contains around 27% occurrences of freelists of size 3, 15% of lists with size 4, and 5% of freelists with size 5. However, as the number of vertices is increased, there is no significant difference in the distributions, thus size is not a crucial factor.

size	% $p=1$	% $p<1$	peak pos	peak %	width
10	14.7	85.3	2	59.3	6
20	14.1	85.9	2	30.8	8
30	30.2	69.8	1	30.2	10
40	9.5	90.5	4	21.3	15
50	23.2	76.8	2	36.5	8
60	21.9	78.1	2	27.3	9
70	22.1	77.9	2	33.7	10
80	29.8	70.2	2	36.2	10
90	13.5	86.5	2	30.8	9
100	21.3	78.7	2	29.5	8

Table 4: Characteristics of freelist distributions when varying graph size.

size	% $p=1$	% $p<1$	peak pos	peak %	width
0.0108	83.5	16.5	1	83.5	3
0.0116	68.3	31.7	1	68.3	4
0.0122	46.4	53.6	1	46.4	8
0.0134	28.2	71.8	2	31.3	11
0.0139	23.2	76.8	2	36.5	8
0.0159	10.1	89.9	3	27.7	10
0.0228	5.9	94.1	2	14.7	22
0.0237	0.7	99.3	6	9.4	29
0.0244	0.4	99.6	12	5.4	70

Figure 5: Characteristics of freelist distributions when varying graph density.

One implication of this data is that even the most trivial of graphs can benefit from a randomized strategy, as only a fraction of the freelists (approximately 20%) has a size of one and is completely deterministic. A freelist with a size of two adds a sufficient level of uncertainty ($p = 0.5$) in making a prediction, for an observer. As the graphs are increased in size the proportion of predictable scenarios (having a freelist size of one) does not change significantly. However, the proportion of freelists with a size greater than two increases, reducing the probability of successful predictions significantly. For example, a size of 3 yields a probability of 0.33 for a successful prediction, a size of 4 yields a probability of 0.25 and so forth.

For the graphs where the graph density was varied, the distributions are strongly affected by the density parameters. Again, the peaks decrease and the distributions become more smeared as the densities are increased. For graphs with a density of 0.0108 the largest freelist has a size of 3, while by increasing the graph density to 0.0244 one obtain freelists with as many as 70 elements. The smearing effect is especially strong as the graph densities approach 1.0 (which generates fully connected graphs). Peaks start at 1 for graphs with a density of 0.0108, 0.0116 and 0.0122. The peak moves to 2 for graphs with a density of 0.0134 and 0.0139, and to 3 for a graph with a density of 0.0159. For the graphs with densities of 0.0228, 0.0237 and 0.0244 distributions peak at 2, 6 and 12 respectively. Further, the proportion of situations where the freelist has a size of one decreases from 83.5% to 4% as the graph density is increased.

Clearly, the density of a graph has a strong effect on the freelist size distributions and thus the random scheduling of these graphs. The more dense a graph, the more random its scheduling can become and thus the more difficult it is for an observer to perform accurate predictions. For large graph densities the probabilities of correct predictions become diminishingly small.

Since the nature of the graph affects the random scheduling, the structure of the precedence relations should be taken into consideration when designing the configuration management topology. The results of this section lead to the following design guidelines.

1. *Size is not important* – the size of the configuration management structure does not significantly affect the effectiveness of the random scheduling strategy, unless the graph is very small, i.e., less than 20 nodes.
2. *Connectivity is good* – graph structures that are strongly connected with many dependencies decrease the probability of making accurate predictions. By increasing the graph density, i.e., the number of edges in the graph, the distribution is smeared outwards. The density of a graph affects the width of the distribution and the size and magnitude of the peak. A high density gives a wide distribution with a low peak situated further away from one.

At first sight these results might seem surprising, especially as the vertices of a graph with zero density (with totally independent nodes) can be scheduled arbitrarily, and a graph with a density of 1 has a fixed scheduling order. However, a graph with a density of 0.0108 is easier to predict than one with a density of 0.0244. The mistake is to assume that a graph with a density of 0.0108 consists of more independent nodes than one with a density of 0.0244. Either the nodes are independent or they are not. If we investigate a ten-node graph with density 0.1, the probability of 0.1 ensures that nearly every row of its adjacency matrix would contain a 1, or an edge. The corresponding

graph would therefore be some loosely connected structure with perhaps one or two independent satellite nodes. In fact, one does not need many 1's in the adjacency matrix to connect the nodes. The result is a narrow and long chain-like structure. A similar argument can be applied when comparing the graphs with densities of 0.9 and 1.

Part III: Malicious Surveillance

In the previous experiment the focus was on the prediction of events and how well one can prevent prediction by scrambling events randomly. The goal of this experiment is to identify how an observer can monitor a system and identify the management structure and configuration patterns. In particular, what is the impact of employing a randomized scheduling strategy on the ability of an observer to identify and reconstruct an accurate model of the configuration management task topology?

In this experiment, it is assumed that the observer is able to monitor the actions of the system administrator, either explicitly or implicitly through observing the results of actions. We also assume that the observer is able to uniquely identify each action.

The example given in the previous experiment can be extended to illustrate this. Assume a user is interested in storing huge files on the system but is aware that there are cleanup configuration actions in place. The user also knows that there are two independent storage areas on the computer system, namely /tmp and /audio/tmp. By writing a simple script that lists the content of these two directories to a file every hour, the user will after one day collect fairly good evidence that /tmp is deleted around 2 am and /audio/tmp around 6 am. Repeating this exercise for several consecutive days confirms the findings. The user has identified the sequence of these two events. This is a trivial example. However, when observing larger number of tasks it is still trivial to identify the patterns as long as the system administrator employs a deterministic strategy that often results in the same repeated sequence of effects. When the same sequence is repeated, the events belonging to the sequence can be modelled using a graph with a linear chain structure, or a total order.

However, if the system administrator employs a random strategy on a graph with a complex topology, the observer can reveal the structure as shown in the following example: A user observes five events over a period of 10 days. Even if the user does not know that there are five events, the user can identify this by observing the recurrence of events. Through the occurrence of the events the user realizes that each event occurs once every day and can therefore deduce that the five events occur in one-day cycles. Each day a sequence of events is observed, for example:

```
01 : 2 1 3 4 5
02 : 1 2 3 4 5
03 : 1 2 3 4 5
04 : 1 2 3 4 5
05 : 2 3 1 4 5
06 : 2 1 3 4 5
07 : 2 1 3 4 5
08 : 2 3 4 1 5
09 : 2 3 4 1 5
10 : 2 1 3 4 5
```

By analyzing these sequences[1] it is possible to deduce the partial ordering of the five tasks. Obviously, tasks 1 and 2 are entry nodes as they are the only tasks occurring in the first position, and task 5 is the only exit task since it is always in the last position. Further, task 3 depends on task 2 since it always occur somewhere after task 2, and task 4 depends on task 3 since it always occur somewhere after task 3. Task five depends on all the other tasks. After, all the transitive dependencies have been removed, one obtains the graph G represented by the adjacency matrix:

$$G = \begin{bmatrix} 0 & 0 & 0 & 1 \\ & 1 & 0 & 1 \\ & & 1 & 1 \\ & & & 1 \end{bmatrix} \qquad (3)$$

A formal procedure for deducing partial orderings from sequences is provided later in this article.

In this small example, the complete structure was uncovered in only a few iterations. However, real world graphs would contain a larger number of vertices. It is possible to make some general statements regarding the time it takes to discover the structure of a graph. If the graph is a linear chain, or a total order, then one observation suffices. However, if the graph consists of N independent vertices, i.e., no ordering, then $N!$ (distinct) observations are necessary in order to establish the fact that all the nodes are independent. As mentioned earlier, the graph usually represents a partial ordering. The number of observations needed to capture the entire structure for large graphs is huge. The few first observations provide a rough indication of the graph topology, and the estimate is refined as further observations are made. However, a large number of observations are needed to uncover all the details.

This discussion is based on the assumption that the reference graph remains constant. However, in real life the management structures are modified on a daily basis to reflect the dynamic needs of the users. Consequently, the graph can be viewed as a time-varying entity. One implication of this is that it is even more difficult for an observer to identify the management structure. However, time varying management structures are beyond the scope of this article.

This experiment sets out to answer how many iterations are necessary in order to identify a configuration management model?

[1]The sequences should be read from left to right.

Method

The experiment was carried out by setting up a sequence *generator* and a sequence *observer*. The generator produced 300 valid random partial orders from each of the graphs in the test suite. The observer observed each of the generated sequences without any knowledge of the graph topology. For each iteration the information embedded in the sequence was accumulated in a process known as *training*. For evaluation and graphing purposes a graph was generated from the accumulated information, all the transitive dependencies were removed and the resulting graph was compared to the reference graph providing the sequences. Each of these steps will be described in the following paragraphs.

A data structure was built up while observing the sequences. The data structure consisted of a list of two sets – a pair of sets for each element, or vertex, in the graph. One of the sets consisted of elements succeeding the current element in the sequence – the *successors*, and the other set of elements preceding the current element in the sequence – the *predecessors*. These sets grew as the observer was introduced to new sequences. This procedure is captured in the following algorithm:

```
for i:=1 to sizeof(S) do
  begin
    suc[i] := union(suc[i], head(S,i-1));
    pro[i] := union(pro[i], tail(S,i+1));
  end
```

S is an array containing the sequence of events. *Sizeof* is a function that returns the size of the sequence, *suc* and *pre* are arrays of sets containing the set of successors and predecessors, *union* is a function returning the union of two sets and *head* and *tail* return sets consisting of the head and the tail of the array respectively.

Equipped with this data structure it is possible to reconstruct the reference graph, either fully or partially, depending on the number of observations made. The graph building algorithm used is based on the following observations:

1. If an element occurs before and after the current element, then the current element and the given element are *independent*. No relationship exist between the two vertices.
2. If an element only occurs before the current element, then the current element depends on that element. The element is a *parent* of the current element.
3. If an element occur only after the current element, then the element depends on the current element. The element is a *child* of the current element.
4. All elements that are independent of all the nodes that may precede them are *entry nodes*. Such elements have no parents.
5. All elements that are independent of all the nodes that succeed them are *exit nodes*. Exit nodes have no children.

The algorithm for extracting the graph is outlined as follows:

```
for i:=1 to n do
  begin
    P[i] := suc[i] - isct(suc[i],pre[i]);
  end
```

P is an array of sets containing the parents of the nodes of the graph, *suc* and *pre* are arrays of sets containing the set of successors and predecessors, and *isct* is a function returning the intersection of two sets. The minus (-) operator removes the elements in the right hand set from the elements of the left hand set.

Transitive dependencies were removed. Graphs free of transitive dependencies were a prerequisite for performing graph comparisons. The reference graph and the predicted graphs were compared as follows: The parents of each vertex in the reference graph was compared to the parents of the corresponding vertex in the predicted graph. The number of common parents where counted and divided by the total number of parents for that vertex (in the reference graph). These ratios were then summed. This procedure be described formally using the following expression:

$$S(G, M(t)) = \sum_{i=1}^{|G|} \frac{P(G_i) \bigcap P(M_i(t))}{|P(G_i)|} \qquad (4)$$

where *S* is the *similarity* function in the range of 0 to 1. A 0 indicates two completely dissimilar graphs and a 1 indicate two completely identical graphs. Further, *G* is the reference graph, *M*(*T*) is the time-variant estimated graph, *P*() is a function returning the set of parents of a node and G_i and $M_i(t)$ refer to node *i* in the respective graphs. It also follows that

$$\lim_{t \to \infty} S(G_i, M_i(t)) = 1. \qquad (5)$$

provided the sequences are generated using a uniformly distributed random variable.

Results

Figures 12, 13, 14 and 15 depict the similarity between the reference graph and the reconstructed graph plotted against time, where time is measured in iterations. The different graphs represent structures with a varying number of vertices.

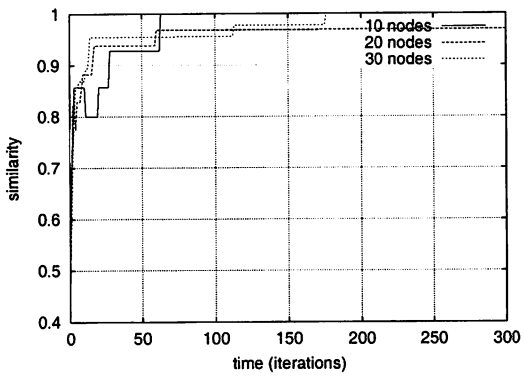


Figure 12: The similarity of the reference graph (10, 20 and 30 vertices) and the predicted graph plotted against time (iterations).

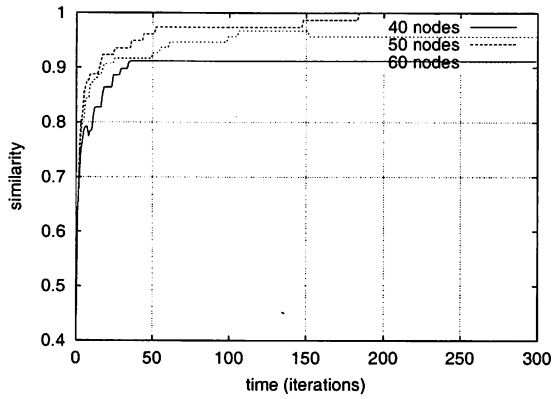


Figure 13: The similarity of the reference graph (40, 50 and 60 vertices) and the predicted graph plotted against time (iterations).

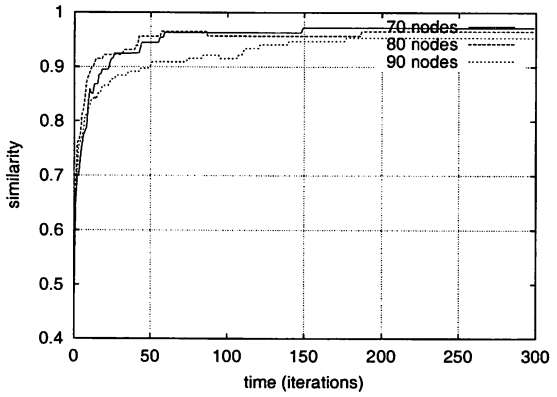


Figure 14: The similarity of the reference graph (70, 80 and 90 vertices) and the predicted graph plotted against time (iterations).

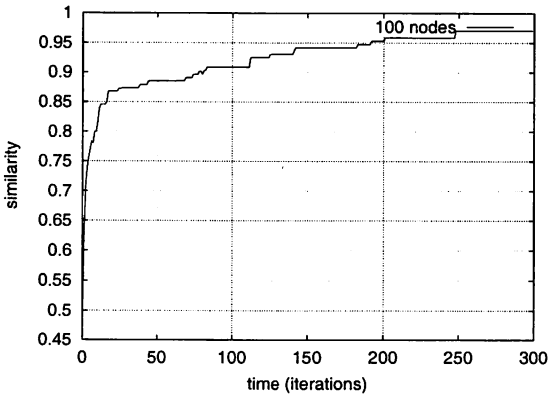


Figure 15: The similarity of the reference graph (100 vertices) and the predicted graph plotted against time (iterations).

Figures 16, 17 and 18 depict the similarity between the reference graph and the reconstructed graph plotted against time, where time is measured in iterations. The different graphs represent structures with a varying graph density.

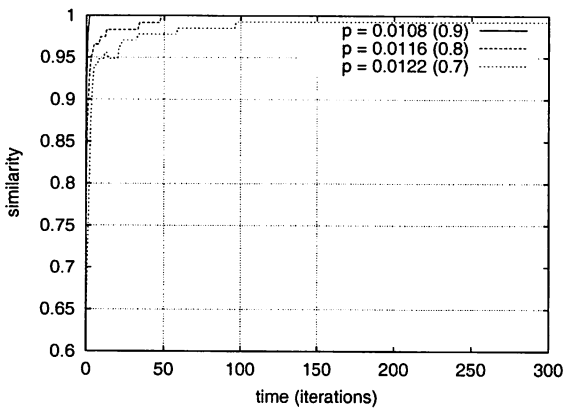


Figure 16: The similarity of the reference graph (densities of 0.0108, 0.0116 and 0.0122) and the predicted graph plotted against time (iterations).

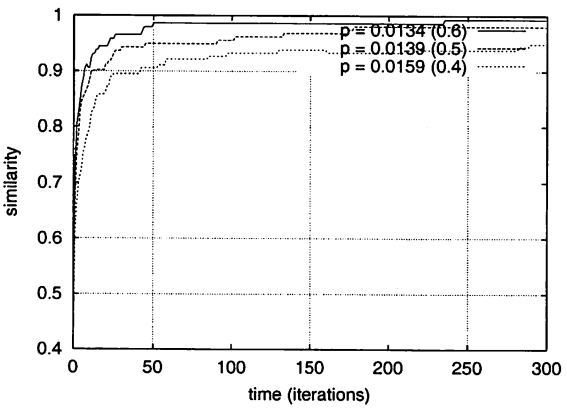


Figure 17: The similarity of the reference graph (densities of 0.0134, 0.0139 and 0.0159) and the predicted graph plotted against time (iterations).

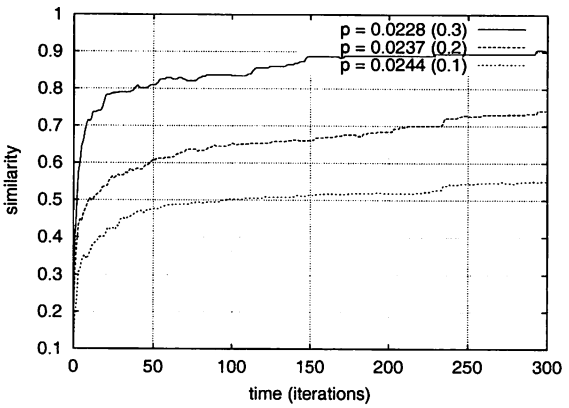


Figure 18: The similarity of the reference graph (densities of 0.0228, 0.0237 and 0.0244) and the predicted graph plotted against time (iterations).

Figures 19 and 20 show graphs of the number of iterations required to reach a similarity levels of 70%, 80%, 90% and 95% for graphs of different sizes and densities respectively.

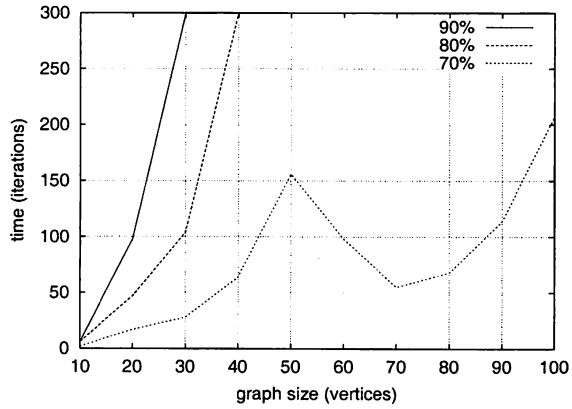


Figure 19: The number of iterations to reach different similarity levels plotted against graph size.

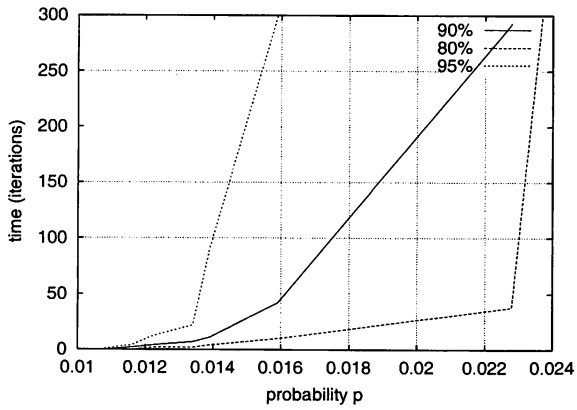


Figure 20: The number of iterations to reach different similarity levels plotted against graph density.

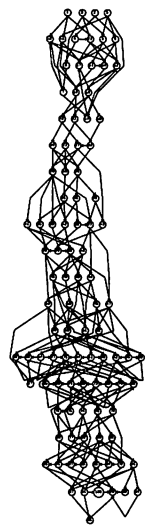


Figure 21: A graph with a similarity factor of 0.67.

Figures 21, 22, 23 and 24 depict graphs at various stages of recognition, namely graphs with similarity levels of 0.67, 0.77, 0.82 and 1.0 respectively. Only, graphs with relatively high similarity factors could be included in this article, as graphs with lower similarity factors had a large depth unsuitable for printed material. These graphs indicate that the depth of the graphs decrease as they become more similar to the reference graph. Similarly the width of the graphs increases as the predicted graphs more similar to the reference graphs. The VCG (Visualizing Compiler Graph) tool was used to generate the graphs [16]. Note that the "minimize depth" option was selected to fit the graphs into these proceedings, as a more natural level by level layout would yield long and narrow graphs.

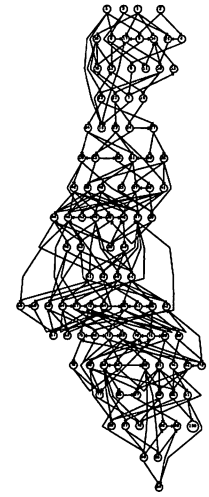


Figure 22: A graph with a similarity factor of 0.77.

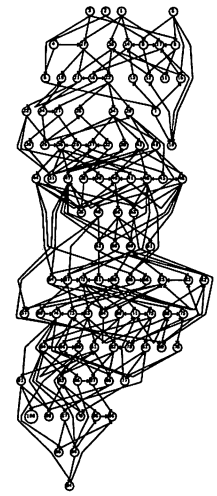


Figure 23: A graph with a similarity factor of 0.82.

Discussion

A common attribute of all the graphs where the similarity level is plotted against time (Figures 12, 13, 14 and 15) is that they appear asymptotic – converging towards 1 or some positive value less than 1. Further, results show that graph size has a moderate effect on obscuring the graph topologies. A larger graph is harder to identify than a smaller graph; more observations are necessary in order to fully identify a large than a small reference graph. This phenomenon is more evident in Figure 19 which plots the number of

observations required to obtain a given similarity level against size. The graphs with 50 nodes, or more, appear to converge onto similarity levels of less than 1, after 300 iterations. This data supports the claim that a randomized strategy obscures the topology of the management structure making it more difficult for an observer to identify it.

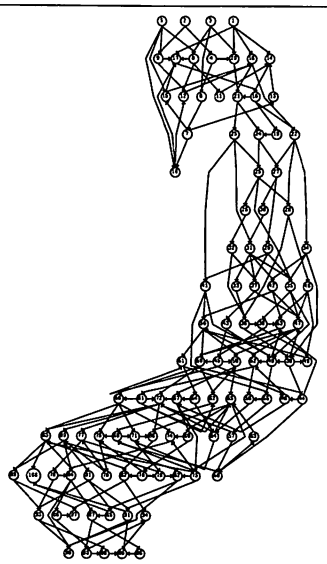


Figure 24: The reference graph.

Graph density appears to have a dramatic and noticeable effect on obscuring graph topologies. For low densities such as 0.0108, 0.0116 and 0.0122 (Figure 16) the observer is able to identify the complete structure or a very similar structure simply after a few iterations. For graphs with medium density, such as 0.0134, 0.0139 and 0.0159 (Figure 17), the similarity graphs converge at significantly lower levels. Finally, the graphs with densities 0.0228, 0.0237 and 0.0244 (Figure 18) are just about starting to converge at the 300th iteration and are still at noticeably low levels of similarity – 0.9, 0.7 and 0.5 respectively. Figure 20 shows the number of iterations to reach the similarity levels of 80%, 90% and 95% are plotted against the graph density, and strongly indicates that the density of a graph has an exponential effect on obscuring the management topology, i.e., with a linear increase in graph density, one achieves an exponential increase in the time required for an observer to reconstruct the structure from the observations.

The visualization of graphs at various stages of recognition (Figures 21, 22, 23 and 24) show that less similar graphs tend to have a larger depth and smaller width than more similar graphs, and that the reference graph itself is the widest and most shallow. This is because initially all nodes are assumed independent, then a few edges are detected, connecting the nodes together in a sparse and deep structure. As more edges are discovered the vertices of the graph are tied more strongly together such that each level becomes wider and the depth becomes more shallow.

Clearly, the experiments indicate that management structures with many nodes are preferable to those with fewer nodes. Further, management topologies with a strong connectivity are drastically less predictable than sparse structures. These observations should be taken into consideration when designing management structures to obscure the management structure to an onlooker.

An Optimal Topology

When given the freedom to design a structure, what topologies yield the strongest connectivity? If one assumes that a graph with N vertices is acyclic and contains no transitive dependencies, then it is obvious that the graph must have a layered structure with L levels, where the d_i nodes of level $i \in L$ are dependent on all the nodes in the previous level $i - 1$ (all other dependencies would be transitive or cyclic). If we assume, for simplicity, that each level contains an equivalent number of elements, i.e., $d_i = d_j$ where $i,j \in L$, such that $dL = N$, then the total number of edges E in the graph is given by

$$E = d^2(L - 1) \qquad (6)$$

Such a graph has a rectangular shaped structure when drawn in a layered manner. Further, E is maximized by maximizing d, i.e, $L = 2$ and $d = N/2$. The resulting graph is a wide structure with two fully connected levels, $E = N^2/4$ edges, and a connectivity of $p = N / 2(N - 1)$. (Note that the transitive edges are not counted).

Conclusions

This paper addresses randomized scheduling of events in a distributed configuration management context. Three, aspects of the impact of randomized scheduling were investigated – namely, efficiency of resource utilization during configuration management, predictability and exploitability by malicious observers, and the extent to which observers are capable of monitoring and identifying the configuration management topology. The experiments show that randomized scheduling has advantages over fixed order strategies – on average resulting in more efficient schedules. Further, most graphs yield sufficiently large freelists that makes the job of predicting management action difficult. Finally, randomized scheduling makes it more difficult for an observer to identify the complete management topology, and if the management topology is viewed as a time-variant entity this difficulty increase even further. When it is difficult to identify the model, then it is also difficult to make accurate predictions. The conclusion to draw from this is that randomized scheduling, in the context of distributed configuration management, can be advantageous compared to a trivial strategy, providing a good compromise between efficiency and security. As random scheduling is an extremely simple strategy to understand and implement it is recommended that it is incorporated into automatic distributed configuration

management tools, such as cfengine. The experiments confirm claims by Burgess [5] that it is advisable to introduce changes into the management model, and continuously change its structure, to increase robustness. Also, as pointed out in [7], randomness can also be introduced in other aspects of the management process such as the timing of the events to improve the robustness.

Acknowledgements

The author would like to acknowledge everyone in the system administrations research group at Oslo University College for enlightening discussions. Further, the author acknowledges Mark Burgess and Alva Couch for insightful and inspirational comments and the anonymous reviewers for their comments greatly enhancing the quality of this manuscript. Finally, the author is grateful to the management at the Oslo University College for recognizing the importance of this work, providing the necessary financial support and a stimulating working environment.

Author Information

Frode Eika Sandnes received the B.Sc. degree in computing science from the University of Newcastle Upon Tyne, England, in 1993, and the Ph.D. degree in computer science from the University of Reading, England, in 1997. He has worked for several years in the space industry developing data handling systems and communications equipment for low earth orbit satellites. He is currently an associate professor in the Department of Computer Science, Faculty of Engineering at Oslo University College, Norway. His research interests include applications of coding theory, signal processing, parallel and distributed processing and especially distributed configuration management. Dr. Sandnes is a member of the IEEE Computer Society. Reach him electronically at Frode-Eika.Sandnes@iu.hio.no|.

Bibliography

[1] Imtiaz Ahmad and Muhammed K. Dhodhi, "Multiprocessor Scheduling in a Genetic Paradigm," *Parallel Computing*, vol 22, pp. 395-406, 1996.

[2] P. Anderson, "Towards a high level machine configuration system," Proceedings of the Eighth Systems, Administration Conference (LISA VIII) USENIX Association, Berkeley, CA, 1994.

[3] Lee Bruno, "Sophisticated Network Management," *Open Computing*, 11(12), p. 100, December, 1994.

[4] M. Burgess, "A Site Configuration Engine," *Computing Systems, MIT Press, Cambridge, MA*, Vol. 8, p.309, 1995.

[5] M. Burgess, "Theoretical System Administration," *Proceedings of the Fourteenth Systems Administration Conference (LISA XIV)*, USENIX Association, Berkeley, CA, p. 1, 2000.

[6] M. Burgess and D. Skipitaris, "Adaptive Locks For Frequently Scheduled Tasks With Unpredictable Runtimes," *Proceedings of the Eleventh Systems Administration Conference (LISA XI)*, USENIX Association: Berkeley, CA, p. 113, 1997.

[7] Mark Burgess and Frode Eika Sandnes, "Predictable Configuration Management in a Randomized Scheduling Framework," in *Proceedings of the 12th International Workshop on Distributed System Operation and Management*, DSOM'2001, INRIA Press, October, 2001.

[8] Alva L. Couch and Noah Daniels, "The Maelstrom: Network Service Debugging via Ineffective Procedures," in *Proceedings of the 15th Systems Administration Conference LISA 2001*, USENIX, December, 2001.

[9] N. Damianou, N. Dulay, E. C. Lupu, and M. Sloman, "Ponder: A Language for Specifying Security and Management Policies for Distributed Systems," *Imperial College Research Report DoC 2000/1*, 2000.

[10] Rik Farrow, "Object-Oriented Network Management," *UNIX/world*, Vol. 9, Num. 11, p. 93, November, 1992.

[11] B. Hagemark and K. Zadeck, "Site: A Language and System for Configuring Many Computers As One Computer Site," *Proceedings of the Workshop on Large Installation Systems Administration III*, USENIX Association, Berkeley, CA, 1989, p. 1, 1989.

[12] B. Jereb and L. Pipan, "Measuring Parallelism in Algorithms," *Microprocessing and Microprogramming, The Euromicro Journal*, Vol 34, pp. 49-52, 1992.

[13] Hironori Kasahara and Seinosuke Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Transactions on Computers*, Vol C-33, Num. 11, pp. 1023-1029, 1984.

[14] S. Omari, R. Boutaba, and O. Cherakaoui, "Policies in SNMPv3-Based Management," *Proceedings of the VI IFIP/IEEE IM Conference on Network Management*, p. 797, 1999.

[15] Raymond Reitter, "Scheduling Parallel Computations," *Journal of the Association for Computing Machinery (ACM)*, Vol. 15, Num. 4, pp. 590-599, 1968.

[16] G. Sander, "Graph Layout Through the VCG Tool," in *Proceedings DIMACS International Workshop GD'94, Lecture Notes in Computer Science 894*, pp. 194-205, Springer Verlag, October, 1995.

[17] Frode Eika Sandnes and G. M. Megson, "Improved Static Multiprocessor Scheduling Using Cyclic Task Graphs: A Genetic Approach," *PARALLEL COMPUTING: Fundamentals, Applications and New Directions, North-Holland*, Vol 12, pp. 703-710, 1998.

[18] Stéphane Schitter, "Integration of Intrusion Detection Products in the Tivoli Enterprise Console," Master's Thesis, Eurécom Institute, June, 1999.

[19] Robert Sedgewick, *Algorithms, Second Edition*, Addison-Wesley, 1989.

[20] Stefan Uelpenich, "Extending the Reach of Tivoli Distributed Monitoring – Creating a Custom Monitoring Collection," *The Managed View*, Vol. 3, Num. 2, pp. 21-40, Spring, 1999.

[21] Herbert Weinblatt, "A New Algorithm for Finding the Simple Cycles of a Finite Directed Graph," *Journal of the Association for Computing Machinery (ACM)*, Vol. 19, Num. 1, pp. 45-56, 1972.

[22] Douglas B. West, *Introduction to Graph Theory Second edition*, Prentice Hall, 2001.

[23] Tao Yang and Cong Fu, "Heuristic Algorithms for Scheduling Iterative Task Computations on Distributed Memory Machines," *IEEE Transactions on Parallel and Distributed Systems,* Vol. 8m, Num. 6, 1997.

[24] M. Zapf, K. Herrmann, K. Geihs, and J. Wolfang, "Decentralized SNMP Management with Mobile Agents," *Proceedings of the VI IFIP/ IEEE IM Conference on Network Management*, p. 623, 1999.

The Maelstrom: Network Service Debugging via "Ineffective Procedures"

Dr. Alva L. Couch – Tufts University
Noah Daniels – Analog Devices

ABSTRACT

The process of network debugging is commonly guided by "decision trees," that describe and attempt to address the most common failure modes. We show that troubleshooting can be made more effective by converting decision trees into suites of "convergent" troubleshooting scripts that do not change network attributes unless these are out of compliance with accepted norms. "Maelstrom" is a tool for managing and coordinating execution of these scripts. Maelstrom exploits convergence of individual scripts to dynamically infer an appropriate execution order for the scripts. It accomplishes this in $O(n^2)$ procedure trials, where n is the number of troubleshooting scripts. This greatly eases adding scripts to a troubleshooting scheme, and thus makes it easier for people to cooperate in producing more exhaustive and effective troubleshooting schemes.

Introduction

In maintaining complicated service networks, one pressing problem is to determine and actively eliminate causes of network service disruptions. It is currently easy to automatically detect network problems through a variety of monitoring techniques [14, 16, 28]. But taking the next step of automatically remedying network problems has so far proven impractical. A human troubleshooter must often engage in involved scientific inquiry to infer 'causes' from observed 'effects.' A complex problem can take weeks to solve, and may be solved without ever revealing the true 'cause' of the problem.

It is currently easy to automatically detect network problems through a variety of monitoring techniques [14, 16, 28]. But taking the next step of automatically remedying network problems has so far proven impractical. A human troubleshooter must often engage in involved scientific inquiry to infer 'causes' from observed 'effects.' A complex problem can take weeks to solve, and may be solved without ever revealing the true 'cause' of the problem.

While it may be argued that a well-designed network and well-chosen hardware do not fail, this is certainly not true in an environment where one teaches hands-on Computer Science. We face problems almost daily with runaway processes, latent bugs in web scripts, and other disruptions based upon student (or faculty) error. Vendor-supplied servers crash due to latent bugs 'discovered' by users toying with new programming techniques. Cables are stepped upon and equipment is abused. We cannot limit the capabilities of users to prevent such failures without compromising our educational mission, so that we must react to failures on an ongoing basis.

One problem with automating troubleshooting of mission-critical network services such as http, ftp, imap, ssh, etc., is that these services typically depend upon other base services (such as NFS, LDAP, database servers, etc.) that must function before the externally visible services will function. Network devices that connect internal servers to one another must also be examined during the troubleshooting process. Ideally, to automate service troubleshooting, one must integrate and coordinate procedures that analyze and repair almost every device in the network, from server processes down to switches. While it may be argued that a network is poorly designed unless the precedences between services are clearly defined and unvarying, problems persist largely because we do not fully recognize or understand the dependencies between services.

Convergence

Troubleshooting is by nature a *convergent process* where one only repairs a component if it seems not to function properly. Monolithic tools such as Cfengine and its relatives [2, 3, 4, 7, 8, 17, 18] attempt to force every attribute of a given device into compliance with a predefined operational policy. Most of these tools are limited to functioning in environments where one can compile and run programs, and cannot be utilized to maintain closed-source vendor components such as routers, switches, dialup servers, and so on. Babble [10] provides the beginnings of a tool for convergent administration of turnkey network services, but is largely limited to conversing with and controlling one device at a time.

Expanding these tools to cover the problem of network-wide troubleshooting seems impractical. They are already limited by their own size and complexity. Some are perhaps reaching the limits of software complexity from the standpoint of usability, predictability, maintainability, and adaptability to new needs.

In this paper, we study the potential for dividing these currently monolithic administrative processes into smaller pieces that work together to accomplish the same goal. Tools designed to be maintained by a small group of experts may require extensive effort from the programmer who wishes to couple new and reusable convergent processes into an existing tool framework. It is more desirable to be able to contribute independent processes that interoperate easily with others without utilizing traditional software coupling mechanisms such as subroutine calls and interfaces.

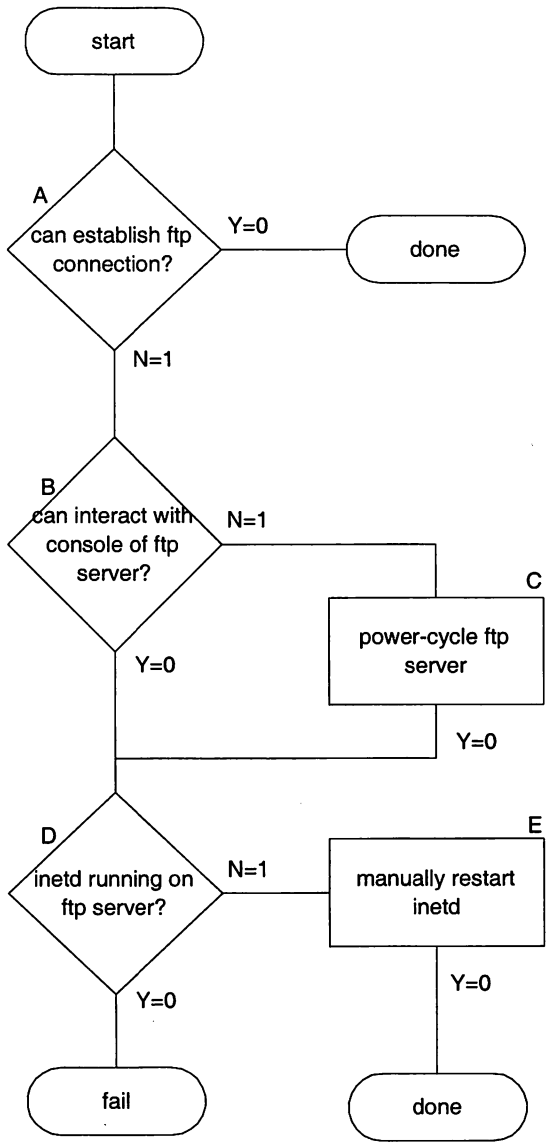


Figure 1: A typical (though oversimplified) troubleshooting decision tree.

Decision trees

As a 'best practice,' many sites pragmatically describe and standardize network troubleshooting procedures as "decision trees" that describe tests to make and corresponding actions to take. Usually decision trees are prepared by senior staff (or vendors) to aid junior staff in dealing effectively and autonomously with common problems. A decision tree is nothing more than a flow chart (in actuality a directed acyclic graph (DAG)) of actions, where the result of each action determines the next action to try.

For an example, consider the (very much oversimplified) decision tree in Figure 1. According to the tree, to check whether ftp service is running, one must first check whether one can make an ftp connection from a client machine. If so, everything is working. If not, one must then check whether the console of the ftp server responds to a keypress. If this succeeds, the server is running; otherwise it has probably crashed and needs to be power-cycled. The final measure is to check whether the program inetd is running on the (now perhaps rebooted) ftp server, and to start it if not. This is far too simple to be realistic; it is contrived only to illustrate concepts rather than as a practical application.

Two special actions determine when to give up. The 'done' action indicates success, while the 'fail' action indicates that the decision tree failed to correct the problem. In a realistic situation, failure of a procedure would escalate the problem's priority and refer it to the next level of technical staff.

One could perform most of the steps in this simple tree by convergent administration in Cfengine [2, 3, 4], but for the purposes of this discussion we consider the steps in the tree to be Babble [10] scripts that potentially interact not just with a local host but also with remote serial consoles of devices such as routers, switches, and power interrupters. While Babble suits our needs for some scripting purposes, these scripts could be arbitrary programs in any desired language.

We started this work with the goal of automating the process of interpreting troubleshooting "decision trees" like the above to automatically detect and repair network disruptions. We abstracted each 'decision' in the tree into a script with multiple exit values 0, 1, 2, ..., where the exit value of a script indicates the next script to invoke. Simple actions not involving a decision are considered to be decisions with only one exit code and outcome. The scripts are plumbed together by declaring which ones should be called as a result of the return codes of others. We implemented a tree traversal algorithm in Perl that "executes the decision tree" by running scripts and taking decision branches as indicated.

Within our scheme, one would implement the above example as scripts A, B, C, D, E, with exit codes determining branches as indicated in Figure 1. For example, if B's exit code is 1, we invoke C; if the code is 0, we invoke D. Using algorithms from Babble [10], we thought we could represent this tree structure in XML [15]. We endeavored to convert this XML tree into a nested hash and execute the results as a kind of script.

Ineffective Procedures

This approach seemed logical, sensible, and straightforward, but failed miserably in practice. Most decision trees for human use are *ineffective procedures*. They suggest what to do, but are typically based upon a simplified view of network function that is understandable to a human. The typical tree addresses a few failure modes and makes assumptions about the precedences between tasks and couplings between components that may not be true in practice.

Perhaps these limitations of a decision tree arise from its real (and perhaps hidden) purpose.

Proposition 1: **A decision tree is not an embodiment of technical knowledge, but rather a statement of operating policy [9].**

It is a description of what a human should "try first," not necessarily what must "work first." The order of tests and actions to be taken are tempered not by physical dependencies, but by service expectations and site mission. For example, 'Rebooting' might be a routine expedient at an academic site, but not at a bank!

Thus one must consider any decision tree not as an effective procedure, but merely a *theory* of what to do when. One does not interpret such a tree literally, but instead bases a course of action upon the information in the tree. In practice, this is what we really do with our own troubleshooting procedures; we 'jump around' within the tree, executing the steps not in order of tree appearance, but in order of their likelihood of causing the problem we have detected.

Exposing Convergent Processes

Most decision trees can be straightforwardly represented as a series of *convergent processes* performed in sequence, each with the purport, "If it is broken, then fix it." A typical convergent process consists of a test and perhaps a configuration change action. If the test succeeds, no action is performed; else an appropriate action is taken in order to ensure that the test succeeds at a later time.

The processes in most decision trees are not convergent in this sense. Sometimes it takes a bit of worthwhile effort to fit an existing decision tree to the convergent process model. For example, one might replace the decision tree above by three convergent processes:

F. Test whether ftp is running, and report success or failure.
G. Test whether the console of the ftp server responds, and power-cycle the server if it seems to have crashed.
H. Test whether inetd is running on the ftp server, and manually restart inetd if it does not seem to be running.

Barriers in the decision tree that prevent unnecessary troubleshooting are no longer necessary. Step F need not fail before one tries steps G and H. None of these steps is damaging to a network that is functioning normally. Step F is a simple test of symptoms that is not even necessary to invoke, because the other two processes are the only 'operative' ones that could potentially change the state of the network and repair a problem.

This convergent representation has advantages over the original decision tree representation. If all three of these processes report success, ftp is running as desired. It no longer matters which process is tried first, as the diagnostic procedures will not take action upon a problem unless the problem has been observed directly and a change is necessary.

In our study we have been unable to generate a network decision tree that cannot be represented as a set of convergent scripts. Network decision trees seldom exhibit more complexity than can be represented straightforwardly in this form. Even when they do, such trees can be expressed in this simpler form by encapsulating more complex processes inside larger "convergent" processes that replace several decisions.

The Curse of Precedence

Transforming a complex decision tree into a sequence of convergent actions is advantageous not only from a human interface standpoint, but also because of the properties of the networks we wish to manage.

The most difficult problem in creating a network debugging decision tree is to determine the precedences between debugging tasks. Many facets of network performance depend upon others. For example, one server may provide NFS service to another that cannot function properly without that service, so one must make NFS work properly before trying to do anything with the dependent server.

Anyone who has tried to manually craft a Makefile for use with the make [23] program will have experienced the complexity of determining and specifying precedences. If the precedences are incorrect or incomplete, make may not have the desired effect, sometimes in subtle ways.

A decision tree is nothing more than a description of the precedences between tasks; in this sense it is a subset of the precedence declarations in a Makefile. But in a network, the precedences between tasks may be constantly changing and are impossible to predefine. As an oversimplified example, consider the problem of a router that utilizes TFTP to boot and then provides routing services for the TFTP host. Suppose we have two convergent processes:

J. Check router function and restart if necessary.
K. Check the TFTP server and restart TFTP if necessary.

In this case, there need not be any fixed precedence between processes J and K; the best precedence depends upon the network's configuration and its current state. If the router is down, then it would be greatly desirable to check the TFTP server first so that

the reboot will work, *unless the* TFTP *server is on the other side of the router from the host running the debugging scripts!* The 'best' ordering for these checks must be determined by considering several factors that may or may not be known:

- The current state of the network and current failure mode.
- Dependencies that apply between components while in that state.

Errors in precedence mean that the overall debugging process will fail, so the success of the process is dependent upon the problem being solved. This means in turn that we must work around this potential problem in a decision tree by encoding *both* orders for the above processes into the tree in some fashion.

In constructing Makefiles, determining the correct precedences is a surmountable problem, because those precedences never change. In network debugging, the problem of determining appropriate precedences between tasks is ill-formed, and there may not be a static precedence between tasks that always applies. Thus make's algorithm of 'topologically sorting' tasks into an execution order coherent with a predetermined partial order is not enough to deal with the precedences between troubleshooting tasks. In fact:

Proposition 2: **Precedence between troubleshooting tasks is an abstract ideal that is not well-defined in practice.**

Exponential Complexity

To generate a decision tree that could succeed in repairing problems in all cases, we would necessarily have to foresee all possible failure modes and encode them into the tree. These failure modes determine the order in which steps should be taken. Each possible ordering or precedence would add complexity to the tree, in the worst case resulting in an exponential explosion in the size of the tree.

These negative conclusions engendered a fundamental change in our thinking and approach. Since we did not believe that we could ever construct such a complex tree correctly, we had no confidence that direct automation of any regular decision tree could *ever* be useful. We stopped even thinking about executing a decision tree directly, and began looking for a better way to approach the problem. The key to this approach is that if a set of scripts exhibit a particular set of properties, precedence does not matter. The key concept is *homogeneity of effect*; that scripts do not conflict with one another in the goals they wish to ensure.

Homogeneity

The concept of a 'convergent script' must be refined slightly when one expects it to interoperate and cooperate with other scripts of similar intent. A script must not only be convergent within itself, but must also behave so that the set of all scripts is convergent as a suite. To do this, each script must exhibit three crucial ingredients:

- **awareness:** a script knows whether it succeeded or not in enforcing its requirements. It returns a nonzero exit code to indicate failure and a zero exit code to indicate success.
- **convergence:** applying the script twice changes nothing and has no effect if the network is already in compliance with the script's requirements.
- **homogeneity** (or *consistency*): scripts never undo the changes made by others (though scripts may enforce the *same* changes as others). This is a *global* convergence criterion over all scripts.

These conditions were of course inspired by the convergent processes of Cfengine [2, 3, 4] and our own Slink [7] and Distr [8], and are exactly the behaviors that we previously attempted to ensure by scripting in Prolog [9] instead of Cfengine or PIKT [24], and in Babble [10] instead of Expect [20].

Of these conditions, the first two are relatively easy to ensure for a given script. These are properties of a script in isolation from others. The third property of homogeneity is a *global condition on the suite of all scripts.* These seemingly abstract ideals are easy to provide in scripts. Consider the simple task of forcing /etc/inetd.conf to have a particular form. A script that does this might contain:[1]

```
cp /proto/inetd.conf /etc/inetd.conf
```

This script can be converted into a convergent one by checking that a copy is needed before making the copy:

```
if [ ! cmp /proto/inetd.conf \
           /etc/inetd.conf ] ; then
  cp /proto/inetd.conf /etc/inetd.conf
fi
```

This can be made aware by arranging for it to return the proper exit codes:

```
if [ ! -f /proto/inetd.conf ] ; \
    exit 1
if [ ! cmp /proto/inetd.conf \
           /etc/inetd.conf ] ; then
  cp /proto/inetd.conf /etc/inetd.conf
fi
exit 0
```

To make it homogeneous with other scripts, however, *every* script that modifies /etc/inetd.conf must modify it into this exact form.

Avoiding Precedence

A set of scripts that exhibit awareness, convergence, and homogeneity with one another has unexpected properties that may not be evident at first glance.

Proposition 3: **Scripts that are convergent and homogeneous may be executed in any order without harm to a functioning network.**

[1] It is generally believed that such a file should be maintained by incremental editing rather than by file copying; this example is oversimplified for clarity.

Since they will not repair non-problems, they will do nothing in a functioning network, and since they will agree on results, they will not undo each others' changes.

This means that even if we do not know the dependencies between scripts, we can dynamically discover an order in which they work properly:

Proposition 4: **Given a set of aware, convergent, and homogeneous scripts that repair parts of a network, we can assure network function by cycling through all possible permutations of the scripts.**

Given a little thought, this claim is relatively obvious. If a script is safe to repeat until it works, and innocuous when not needed, one can simply try it in all possible contexts until it works. If there is an order in which the scripts will work, that order will be tried, so that precedences will be satisfied.

In our decision tree example, step H depends upon the success of step G, while step F depends upon the success of step H, so that the appropriate execution order is "GHF". But even if we do not know this order, we can still utilize the tests effectively by repeating them so that all possible orders will be contained in the pattern of repetition. If we execute the steps in the order "FGHFGHF", the appropriate "GHF" subsequence is present in that ordering. The sequence "FGHFGHF" contains *all possible permutations* of F,G, and H as subsequences:

```
(FGHFGHF)
 FGH....
 F.H.G..
 .GHF...
 .G.F.H.
 ..HFG..
 ..H.G.F
```

After this sequence of executions, ftp will be running if it possibly can be made to run by the debugging processes as given.

Trying All Permutations

Given any set of scripts that are aware, convergent, and homogeneous, one can try them in all possible permutations in a very straightforward way. One exploits the following mathematical fact:

Proposition 5: **Given a sequence $S = A_1 \cdots A_n$ of n objects, the sequence containing $n-1$ copies of S followed by the first object A_1 in S contains all permutations of the members of A as subsequences. This sequence contains $n(n-1)+1$ elements.**

Counter to intuition, enumerating all permutations as *subsequences* of a given sequence does not require $O(n!)$ steps, but only $O(n^2)$ steps.

As an aside, this fact is easy to see by inductive proof. The inductive basis case is that for two objects A_1 and A_2, the appropriate sequence is $A_1A_2A_1$. If we presume that the proposition is true for $A_1 \cdots A_{n-1}$, then a sequence S_{n-1} consisting of $n-2$ copies of $A_1 \cdots A_{n-1}$, followed by A_1, contains all permutations of $A_1 \cdots A_{n-1}$ as subsequences. For A_n elements, we construct the sequence S_n of $n-1$ copies of $A_1 \cdots A_n$, followed by A_1. Our claim is that S_n contains all permutations of $A_1 \cdots A_n$ as subsequences.

To show this, we choose one element A_i from the first copy of $A_1 \cdots A_n$, and consider the subsequence of this sequence constructed by starting after A_i and deleting A_i from the rest. The beginning of this subsequence always has the form of the sequence in the inductive hypothesis, and thus contains all permutations of the A's without A_i. Thus the sequence S_n (of which this is a subsequence) contains all permutations *starting* with A_i. As A_i was arbitrary, S_n contains all permutations of the A's.

For example, consider what happens with four scripts A, B, C, and D. If we start with the sequence ABCDABCDABCDA and select, e.g., C from the first group, the subsequence constructed according to the proof is ABDABDA, which has already been shown to contain all permutations of A, B, and D. By repeating this process we can generate all permutations of A, B, C, and D:

```
subsequence      permutations
 ABCDABCDABCDA
 ABCD.BCD.B...  (A first)
 .BCD..C.....   ABCD, ABDC
 ..C..B.D.B...  ACBD, ACDB
 ...D.BC..B...  ADBC, ADCB
 .BCDA.CDA.C..  (B first)
 ..CDA..D.....  BCDA, BCAD
 ...DA.C.A....  BDAC, BDCA
 ....A.CD..C..  BACD, BADC
 ..CDAB.DAB.D.  (C first)
 ...DABA......  CDAB, CDBA
 ....AB.D.B...  CABD, CADB
 .....B..A..DA   CBAD, CBDA
 ...DABC.ABC.A   (D first)
 ....ABC..B...  DABC, DACB
 .....BC.A.C..  DBCA, DBAC
 ......C.AB..A   DCAB, DCBA
```

This is not the best known solution to the problem of "Permutation Embedding." For n objects there is a sequence of $n^2 - 2n + 4$ elements that contains all permutations of the original objects as subsequences [1, 5, 13, 19, 22, 26, 27]. The methods utilized to construct a sequence with this lower bound are less intuitive than ours, while the size of the sequence remains $O(n^2)$ in all cases. To our knowledge, finding the optimal solution in the general case remains an open problem in combinatorics.

In our case, the objects being permuted are scripts that manage the values of configurable attributes, while the sequence is the execution order for the scripts. If we know that a correct order for the scripts exists, then if we try to execute them in the order suggested above, that order will be achieved at some time during the process. Because of our homogeneity constraint, each script need work only once

and scripts will not undo the work of others, so that one iteration of the scripts in the proper order is all that is required. Thus the precedence constraints for all scripts will be satisfied if there is a way to satisfy them at all, and *the algorithm dynamically infers the correct execution order for the scripts as it executes.*

Exploiting Awareness

If we know that the scripts we are running are aware as well as homogeneous, we can make this iterative process more efficient. If they are aware, they know when they succeed. If they are homogeneous, one success for each script suffices. Thus we need only execute each script until it succeeds, and the permutation embedding algorithm can skip invoking scripts that have already succeeded.

Suppose, e.g., that we have six scripts A,B,C,D,E,F, to be executed in that order, and that in actuality D and E must occur before B; E before C; C before A; and A before F. Let us notate a success of A as =A (representing unification with A's postconditions) and a failure of A as !A. Then the algorithm's execution will proceed as follows:

```
!A !B !C =D =E !F
!A =B =C .. .. !F
=A .. .. .. .. =F
```

In the first pass, A, B, C, and F fail, while D and E succeed, because !C implies !A, !D and !E imply !B, !E implies !C, and !A implies !F. In the second pass B and C succeed because their predecessors D and E have succeeded in the first. In the last pass, A and F succeed because all their preconditions are satisfied. So an appropriate total order is D, E, B, C, A, F.

The worst case is that the precedences are the reverse of the order of the list. In this case the algorithm takes $O(n^2)$ executions, where n is the number of scripts:

```
!A !B !C !D !E =F
!A !B !C !D =E ..
!A !B !C =D .. ..
!A !B =C .. .. ..
!A =B .. .. .. ..
=A
```

There are $(n-1)(n-2)/2$ script failures due to inappropriate precedence. Once we know the reverse order is more appropriate, however, a subsequent run will take only $O(n)$ executions.

The driving idea here is that

Proposition 6: **The execution order for an aware, convergent, and homogeneous set of scripts need not be predeclared through precedences, but can be discovered by executing the scripts and observing their effects.**

We should note that this does not mean that we can discover the true precedences between scripts, just that we can discover *some execution order that obeys those precedences.* Further, Frode Eika Sandnes shows that knowing this order does not in general aid in inferring the true precedences between actions [25].

The Maelstrom

Maelstrom is a tool that implements the above algorithm for dispatching debugging scripts. The input to Maelstrom is a configuration file that describes the *probable* precedences between a collection of debugging scripts. These need not be the actual precedences between scripts; they represent just a *suggestion* of the best order in which to execute the scripts. Maelstrom begins by ordering these scripts into a total order consistent with the partial order declared in the file, in the manner of make [23].

The scripts that Maelstrom controls are expected to know whether they succeed or fail. A script tells Maelstrom that it succeeded in its task by returning an exit code of 0. Maelstrom interprets such a response as an indication that the script either found the network in compliance with the script's requirements or made the network comply by making changes. Maelstrom makes no distinction between these cases. If a script returns a nonzero exit code, Maelstrom interprets this as an indication that the script could not succeed for some external reason. Such scripts are tried again at a later time to determine if other scripts make it possible for them to succeed.

Maelstrom attempts to make all the scripts succeed – indicated by an exit code of 0 – by trying a sufficient number of permutations of the scripts. Given n scripts, each script is tried at most n times. When a script succeeds, it is not tried again. If precedences are correct, each script is executed exactly once and succeeds immediately, while if precedences are completely backward, each script is executed n times with $n-1$ failures preceding one success. At the end of trying all possible permutations of the scripts, Maelstrom gives up and reports the scripts that failed consistently. If this happens, then the scripts have failed to repair the network and some other script (or human intervention) is required in order to repair the problem.

Crafting the Perfect Storm

For convenience and ease of use, Maelstrom's configuration file looks much like a Makefile. The directive:

```
c1 : c2 : c3
```

describes the precedences between scripts c1, c2, and c3: c3 before c2 before c1. The difference between this and a Makefile is that there are nothing but scripts here; there are no other files or intermediary results listed. Scripts can have arguments. Two script invocations are considered identical if they have the exact same arguments, so that a script must succeed once with each set of arguments. Thus the declaration:

```
c1 -f : c2 --tftp
c2 --tftp : c2 --comsat
```

describes the relationships between *three* scripts, "c1 -f", "c2 --comsat" and "c2 --tftp".

Commands in Maelstrom are repeatedly executed until they succeed, as indicated by a return code

(exit status) of 0. If they do not succeed, they are repeated in sequence according to the total order suggested by a topological sort of the partial order described by the file. During this process, Maelstrom skips any scripts that previously succeeded during the repetition. In the last example above, the total order is:

A. c2 --comsat
B. c2 --tftp
C. c1 -f

Maelstrom would execute these in the order "ABCABCA," which as above contains all permutations of A, B, and C as subsequences.

Maelstrom and make

It should be obvious by now that Maelstrom uses a very different execution algorithm than make. First, Maelstrom presumes that all of the scripts it controls use exit codes to indicate whether they should be repeated. In make, a non-zero exit code instead indicates an irrecoverable error and causes a full stop. The most crucial difference, though, is that make utilizes centralized or *global knowledge* of precedence in order to schedule its tasks, while Maelstrom's knowledge is a local result of the behaviors of the scripts it controls. This means that to emulate make with Maelstrom, some of the tasks done by make must move into the scripts that Maelstrom dispatches.

For example, consider the trivial Makefile:

```
foo: foo.o bar.o
        g++ -o foo foo.o bar.o
foo.o: foo.c
        g++ -c foo.c
bar.o: bar.c
        g++ -c bar.c
```

This describes how to make an executable foo from source files foo.c and bar.c. To simulate this in Maelstrom, we break it into three convergent tasks c1, c2, and c3: one for each compilation command in the Makefile. The convergent process of creating foo.o from foo.c is easy to express in the (shell) script c2:

```
if [ -nt foo.o foo.c ]; \
    exit 0
g++ -c foo.c
exit 0
```

(-nt x y is true if x is newer than y). Likewise, compiling bar.c into bar.o can be accomplished in a script c3:

```
if [ -nt bar.o bar.c ]; \
     exit 0
g++ -c bar.c
exit 0
```

Difficulties, arise, however, in performing the final step of creating foo from foo.o and bar.o. In make, the states and dates of all files are known, and this global knowledge is used to assure that each command operates upon up-to-date data. Maelstrom knows none of this, so we must compensate for its lack of awareness by checking all prerequisites inside the compilation script before doing the final compilation:

```
if [ -nt foo.c foo.o ]; \
  exit 1
if [ -nt bar.c bar.o ]; \
  exit 1
if [ -nt foo foo.o \
  -a -nt foo bar.o ]; \
  exit 0
g++ -o foo foo.o bar.o
exit 0;
```

The script that combines all results must compensate for Maelstrom's lack of global knowledge by obtaining that knowledge for itself. If we call this script c1, then the Maelstrom configuration:

```
c1 : c2
c1 : c3
```

has the same result as the above Makefile.

This may seem like a limitation until we realize that in Maelstrom's problem domain, the kind of global knowledge that make utilizes is typically unavailable or approximate. Since troubleshooting scripts, for safety, *must* check that their prerequisites are satisfied before continuing, Maelstrom's lack of global knowledge does not impact its ability to solve troubleshooting problems.

Implementing Policy

Sometimes decision trees are effective procedures. Thus we have made it possible to implement traditional decision trees (such as the one at the beginning of this paper) using Maelstrom's syntax. Decision trees are implemented by specifying execution rules for scripts based upon the exit codes of others.

Three forms of policy control in Maelstrom allow forcing execution of specific scripts immediately after the failures or successes of others. Forced evaluation is a policy decision based upon what is most important in a network. One might wish, e.g., to minimize the impacts of a reboot by immediately checking specific services related to the reboot, before checking other facets of operation. Unlike the core scripts Maelstrom invokes, these are *not* interpreted as suggestions or theory, but directly control what Maelstrom does in specific situations.

All the execution controls mimic shell syntax for ease of use. Short-circuit 'and' and 'or' work as they do in the shell:

```
A || B
```

causes script B to be invoked only if script *A* fails. Likewise,

```
A && B
```

causes script B to be invoked only if script *A* succeeds. Equivalently,

```
!A && B
```

has the obvious meaning. An advanced syntax allows reacting directly to exit codes:

```
A [ 23=>B; 34=>C ]
```

means that if the exit code of A is 23, execute B, and if the exit code of A is 34, execute C.

Primary and Secondary Scripts

Scripts that are called as the result of deterministic rules are treated differently by Maelstrom than the scripts to which such a rule applies.

- *Primary* scripts are those that appear alone, in precedence declarations, or on the left-hand side of the above conditional execution forms (e.g., A in A||B or A&&B or A[B;C]).
- *Secondary* (or incidental) scripts do not appear in any of the primary contexts.

Maelstrom attempts to make all the primary scripts in its configuration file succeed once. Secondary scripts are not included in Maelstrom's algorithm, and only execute under the conditions in which they are declared. This allows one to code traditional decision trees into Maelstrom processes without invoking the Maelstrom algorithm upon their components.

For example, the tree at the beginning of this paper, containing scripts A, B, C, D, E could be coded explicitly as:

```
A[1=>B[1=>C;0=>D[1=>E]]]
```

or by realizing that X[1=>Y] is just X||Y, more simply as:

```
A||B[1=>C;0=>D||E]
```

In this case:

- A is a primary command and becomes part of Maelstrom's main task.
- B, C, D, E are secondary scripts that are only invoked when A fails. They are not part of Maelstrom's list of scripts that *must* succeed.

Using the [] syntax, very complex trees are possible, though for simplicity any one script invocation can be assigned only one set of actions. Writing the two declarations:

```
A || B
A [ 1=>C ; 2=>D ]
```

together is illegal, because there can only be one response to an invocation of A.

A Maelstrom command is parsed by first splitting it into phrases separated by the precedence operator ':'. These phrases consist of a list of commands separated by ';'. Each command can optionally be associated with an action by use of the decision operators '&&', '||', or '[]', where the latter can themselves contain lists of commands and associated actions. A decision can appear after a command in any context, e.g.,

```
A [ B [ C ; D ] ; E || F ] : G && H
```

is precisely equivalent to

```
A : G
A [ B [ C ; D ] ; E || F ]
G && H
```

and roughly equivalent to:

```
A : G
A [ B ; E ]
B [ C ; D ]
E || F
G && H
```

In the first two declarations there are two primary commands A and G, while in the last one the primary commands also include B and E (because they appear outside the context of a conditional command).

Testing the Wind

Maelstrom was much more difficult to test than to write, because few of us would enjoy releasing a tool of such powers to do damage to our network during testing! Thus testing required writing a simulator whose input was a known set of task precedences, to see if Maelstrom could sort them out without foreknowledge of their precedences. It does this as expected.

Quite obviously, the success of Maelstrom depends upon the quality and reliability of the scripts it dispatches. To date, we have not deployed Maelstrom in production, because we are not yet confident of our scripts' convergent properties. The scripts that we envision using in production are largely targeted at restoring specific mission-critical services, and not at addressing systemic failures or network connectivity as yet. A typical script tries a netcat from a remote device to see if a service is working *from outside a server* and then attempts to repair any problems *inside* the server, as is possible by using Babble.

Safety

Even in the simple examples of this paper, there are conditions in which a troubleshooting script can make things worse by its actions. This can happen if a corrective action is too extreme or depends upon external resources that are themselves down at the moment. If our scripts are convergent in the Cfengine sense and there are no hidden constraints, Maelstrom is relatively safe. If, e.g., a reboot depends upon hidden constraints that have not been assured, such as a service required for reboot, Maelstrom may reboot a server even though this makes the network state worse, and may well make future troubleshooting impossible without operator intervention.

Maelstrom is currently relatively naive about the limitations of its environment. It can be made safer by giving it more understanding of the imperfections within its scripts, and the hidden couplings between scripts and Maelstrom's environment.

Maelstrom cannot currently compensate for inhomogeneity or lack of convergence in scripts. In the future, there will be stronger precedence operators to control *Maelstrom's* actions in the presence of imperfect scripts. Recall that "c2 : c1" means "c1 might theoretically precede c2." We plan other precedence operations whose main purpose is to compensate for script deficiencies:

- "c2 :: c1" will mean "the *last success* of c1 must precede the *last* invocation of c2."
- "c2 ::: c1" will mean "the *first success* of c1 must precede the *first* invocation of c2."

Both of these are still weaker conditions than the ":" in make. In Maelstrom, we could notate make's concept of strong precedence as follows:

- "c2 :::: c1' could mean that *every success* of c1 must be followed by an invocation (and success) of c2.

We use the colons to limit the number of characters one must escape inside shell commands in the configuration file (currently ':', ';', '[', and ']').

All of these syntactic mechanisms are attempts to compensate for non-homogeneous or non-convergent behavior in scripts.

- The declaration c2 :: c1 means that c1 and c2 are convergent but inhomogeneous, so that c2 must be tried after c1 in order to make the execution result deterministic. This rule means "always clean up after c1 with c2."
- The declaration c2 ::: c1 declares a hard-coded precedence, and means "it is impractical to execute c2 without at least one success of c1. We will use this when there are unavoidable physical dependencies, such as when an intervening router must be tested before the equipment behind it.
- The declaration c2 :::: c1 (which we may *not* implement in the immediate future) compensates for *non-convergent* behavior of c1, by *always* following it as soon as possible with a cleanup routine c2.

To understand the importance of adding these precedence operators to Maelstrom, note that with even the first one (::) we can simulate make with Maelstrom without resorting to more script intelligence. If script c1 is:

```
if [ -nt foo foo.o \
  -a -nt foo bar.o] ; exit 1
g++ -o foo foo.o bar.o exit 0
```

and script c2 is:

```
if [ -nt foo.o foo.c ] ; \
  exit 0
g++ -c foo.c
exit 0
```

and script c3 is:

```
if [ -nt bar.o bar.c ] ; \
  exit 0
g++ -c bar.c
exit 0
```

then the Maelstrom declarations:

```
c1 :: c2
c1 :: c3
```

would accomplish the same *effect* as the Makefile above. Even the relatively weak double-colon operator precedence avoids the need to have script c1 know all the dependencies between its files, as in the former example. This script *might* do redundant compilations, but in the end it will accomplish the exact same result as the Makefile.

Although we discuss the possibility of 'rebooting' as a result of a script, we are not happy with the prospect of automated power-cycling of servers. We are currently developing a tool that allows that kind of dangerous action to be controlled by an electronic mail or two-way pager transaction. The script that wishes to reboot a server asks us whether it should or not, and an operator can mail back a 'yes' or 'no' response.

One weakness of Maelstrom's scheduling is its simplicity. Many colleagues have suggested that Maelstrom should allow one to declare not just precedences, but also "costs" as a measure of how disruptive a particular action will be. One could then try solutions in order of increasing cost. But this would require an even more complex syntax in the configuration file, and theoretical precedences have the same overall effect (through different kinds of declarations).

Observability

The Maelstrom framework allows one to 'glue together' several scripts toward a common goal. Precedences do not have to be specified during the gluing because Maelstrom will sort the scripts into an appropriate order during execution. During the sorting process, Maelstrom *does not determine the actual precedences between scripts, but rather a total order that satisfies those precedences.* The order is observable but the precedences are not [25]. Similarly,

Proposition 7: **While the *effects* of homogeneity are observable, homogeneity itself is an abstract ideal that is unobservable in practice.**

We will illustrate this fact with several examples.

For simplicity, let us consider the case in which all scripts control configuration attributes that have static values in the absence of script effects. This ignores dynamic attributes that can change on their own without script intervention. If the former are problematic, the latter surely are as well.

In this simplified model, there are only *attributes* to be controlled and *parameters* describing ideal settings. A *configuration attribute* is any such data, including text or numeric fields in configuration files or device memory, or even hierarchies of configuration data as handled by Arusha [17] or Babble [10]. A particular script can perhaps do two things with an attribute:

- *read* its value to validate this value against established norms.
- *write* a value into the attribute in order to put it into compliance with such norms.

The value of a particular attribute can be compared with a *configuration parameter* that describes its ideal state. This can either be a fixed value or a rule that describes the ideal value in a variety of situations.

Using this simplified model of script behavior, we can describe correct and compliant script behavior as well as possible deviant script behaviors:

- A script that is aware can read all the configuration attributes that it sets to check for success or failure.
- A script that is convergent only changes attributes that are "out of sync" with desired parameter settings. This means that it reads every attribute and writes it only if it does not satisfy parameter requirements or rules.
- A script that is homogeneous agrees with other scripts upon parameter values. This can be accomplished either by synchronizing the contents of multiple parameter sources or by depending upon a single source for parameter and policy information.

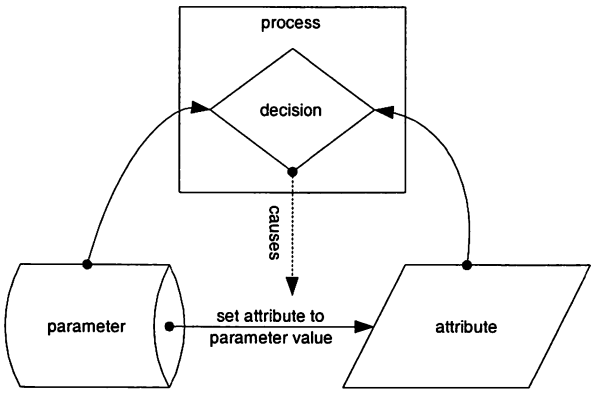


Figure 2: A healthy transaction copies a parameter into an attribute only if needed.

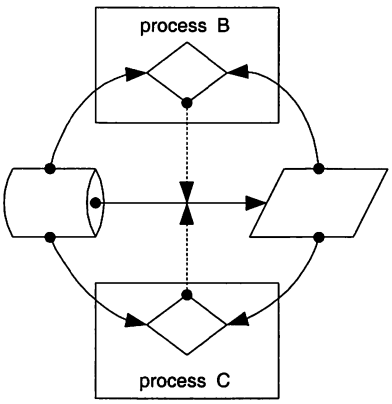


Figure 3: Two homogeneous scripts using the same parameter source.

Graphically, the relationships between scripts, attributes, and parameters may be depicted by representing scripts as rectangles, decisions as diamonds, parameters (or policies) as storage drums, and device attributes to be controlled as parallelograms. An arrow between two objects means that the first affects the second, even when one or more of the objects being affected are themselves arrows. An arrow between a parameter and an attribute means that this parameter affects that attribute by setting it to a compliant value. An arrow from a script to the prior *arrow* means that the script in question enforces that relationship.

For example, in Figure 2, we have one healthy configuration process that controls one attribute with respect to one parameter. The process reads both the attribute and its controlling parameter, and the attribute is only set if needed. This is indicated by a decision node whose inputs are the values to compare, and which controls the (data flow) arrow between the parameter and the attribute in question.

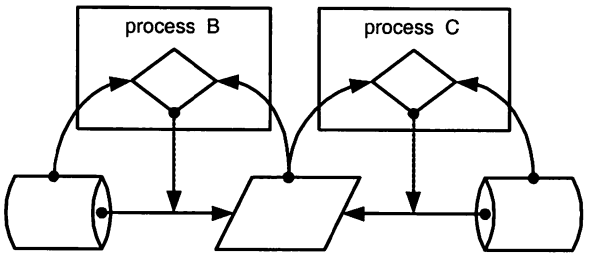


Figure 4: Two inhomogeneous scripts using different parameter sources.

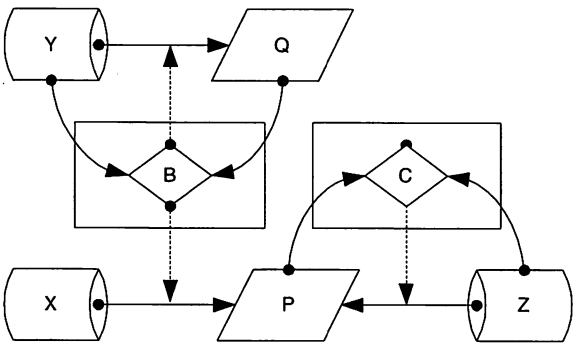


Figure 5: Latent inhomogeneity masked by a single environmental side-effect.

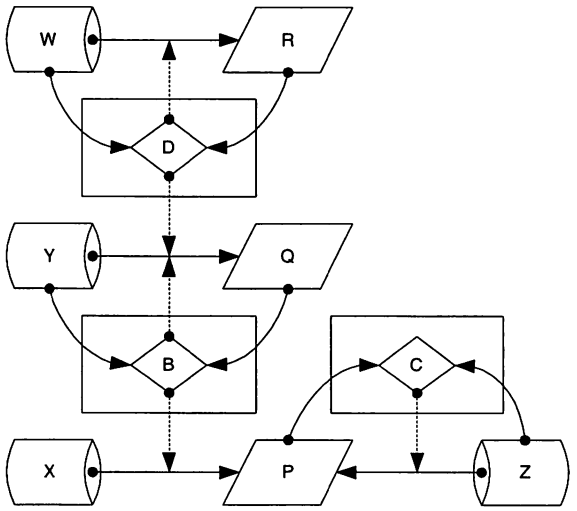


Figure 6: Latent inhomogeneity hidden by multiple environmental side-effects.

This notation allows us to notate many things that can go wrong (or right) when scripts try to cooperate. Homogeneity of scripts means that when they control the same attribute, they utilize the same source

(Figure 3). Inhomogeneity results from two processes that try to force the same attribute into conformance with conflicting parameters or policies (Figure 4).

It would seem from the above example that inhomogeneity is easy to observe and detect, but it can be easily masked and obscured by relatively common scripting errors so that it is only observable under specific environmental conditions.

For example, in Figure 5, script B changes an attribute P to a parameter value X depending upon conformance of a distinct attribute Q to a parameter Y. If another script C sets P according to another parameter rule Z, then the result of the sequence BC will always give P the value from Z, while the sequence CB will give P an inhomogeneous value *only if* Q *is initially out of conformance with* Y. Thus inhomogeneity of B and C *is not observable* except under certain (typically unknown) environmental preconditions.

One might think that this condition would be rare in traditional practice, but we actually find it very common in *our own* scripts! For example, we might want to update someone's password on a machine or network. This password is stored, with other information – such as the user's print name – that might be changed inadvertently at the same time as the password is changed.

Side-effects can make inhomogeneity arbitrarily difficult to detect. In Figure 6, three processes B, C, and D have homogeneous effects except when parameter W matches attribute R, parameter Y does not match attribute Q, and B and C are executed after script D. In this case alone, scripts B and C are inhomogeneous and the orders DBC and DCB will set attribute P according to conflicting parameters Z and X, respectively.

Thus inhomogeneity is very difficult to detect in general, even when the scripts being tested work properly in isolation and are fully convergent and aware of their effects. Using a generalization of the above construction, one can create a set of scripts for which every permutation produces the same result except within one particular environment, wherein two permutations produce inconsistent results.

Assuring Determinism

So far, whenever we have been unable to predefine something we need, we have consistently tried to observe it instead. When observing, we found that *the original abstract concept that we tried to describe is not what we need, and that something less descriptive and more observable suffices.* While we initially searched for a way to define precedences for tasks, we only truly need an order in which the tasks will function properly. This order can be discovered dynamically, even if the precedences remain hidden.

Applying this approach to homogeneity of a set of scripts, it is the observable results of homogeneity that we desire, rather than the property itself. Minimally, the execution of every set of scripts should have a deterministic outcome so there are 'no surprises.' If this outcome is inappropriate, we know that our scripts are incorrect. If a script has an inhomogeneity that does not affect the outcome, we will never know about it, and perhaps need not even care.

To assure determinism in the outcome of executing a set of scripts that are convergent and deterministic in isolation from one another, we can run them in a particular order on a *functioning* network. If they are homogeneous with one another, this will make no changes to the network. If they are inhomogeneous but convergent, a script that disagrees with another will make a conflicting change to the network. Even so, *the fact that the scripts were executed in a predetermined order guarantees a result dependent upon that order.* Once we have agreed upon this as a goal, detecting inhomogeneity becomes easier. If each script reports whether it changed anything, we can record whether there are any inhomogeneities apparent within the sequence in which we execute the scripts. This is much easier to determine than detecting inhomogeneities between all pairs of scripts, as we need only concern ourselves with anomalies in executing *one permutation* of the scripts, not all permutations (as would be required to detect the subtle examples of inhomogeneity presented above).

Suppose that in a *functioning* network, we run a sequence of scripts *twice*. We can conclude, in the absence of other external effects, that any script that reports changes twice during this process is inhomogeneous with at least one of the scripts invoked between its two invocations. For example, if the sequence of scripts is ABCDE, and we invoke them twice with the following results:

```
script: A B C D E A B C D E
change: n y n n y n y n n y
```

we can conclude that B and E are inhomogeneous with one another, while the other scripts are observably homogeneous. In more complex cases it may not be clear which scripts are inhomogeneous with others, as in

```
script: A B C D E A B C D E
change: n y y n y n y n n y
```

We can say little about the relationship between C and the others, as it only makes changes *once*. All that we can conclude from this is that E must follow B to ensure the result achieved by this run. It is unclear whether C is inhomogeneous with either B or E from this evidence.

Coping with Inhomogeneity

Just as homogeneity guarantees that the effect of a sequence of scripts is independent of its order, lack of homogeneity means that the effect of a sequence of scripts depends somewhat upon their ordering. While repeating the scripts in a functioning network in a predetermined order assures a deterministic result, we can restore some semblance of predictability with less

effort by restricting ourselves to a subset of execution orders for the scripts during the troubleshooting process.

For example, consider scripts A, B, C, and D. Suppose that A is homogeneous with all others, D is homogeneous with all others, and B and C are not homogeneous with one another. This is a precise way of saying that BC (B success followed by C success) may perhaps exhibit a different result than CB. Thus there are two groups of permutations of the set of scripts that produce two distinct results:

result 1	result 2
BCAD	CBAD
BACD	CABD
BADC	CADB
ABCD	ACBD
ABDC	ACDB
ADBC	ADCB
BCDA	CBDA
BDCA	CDBA
BDAC	CDAB
DBCA	DCBA
DBAC	DCAB
DABC	DACB

Suppose that we wish to decide (perhaps arbitrarily) upon BC. Then we must execute C after B even if C has already succeeded. This is a less extreme solution than executing all of the scripts again in a functioning network, but in this case has the same effect. In the future, we will be able to assure this kind of behavior in Maelstrom by using extended precedence operators ::, :::, and perhaps ::::.

Latent Variables

In practice, however, homogeneity can be much more difficult to identify or refute, because other problems and processes can cause the same symptoms. Suppose for two otherwise homogeneous scripts B and C, there is *yet another unknown rogue process* interfering with B. In this case, the fact that B had to correct a problem twice does not indicate a sequencing problem with C, but is rather due to a hidden latent variable that has nothing to do with the performance of either script we are considering. Conversely, a successful test like the above can only indicate that homogeneity is preserved in one test case. Any such analysis only suggests that homogeneity *may* or *may* not be preserved, but does not indicate whether it *is* or *is not* preserved. Sadly, the latter are mathematical ideals that are meaningless in practice.

Phenomenology

In the above arguments, twice we have applied observation as a replacement for description, first in determining the order of troubleshooting scripts, and then in determining whether a particular set of scripts is capable of being ordered in that manner. These are both 'phenomenological' (or 'empirical') approaches that replace abstract descriptions with hard evidence, by *changing the problem to allow employing evidence instead of theory.*

In traditional approaches to troubleshooting, theory guides all actions. In the context of this paper, the precedences between tasks are theoretical. Interpreting a valid theory literally improves speed of troubleshooting by avoiding redundant tests, while an invalid theory may blind one to a potential problem by keeping one from executing tests in the order required by physical conditions.

Modularity

The rules above for scripts allow scripts to be added freely to our debugging schema without any attention to precedences between scripts as long as they all remain homogeneous, convergent, and aware of their effects. These requirements allow a new form of 'phenomenological' software modularity; a modularity based upon effect rather than interface. If scripts can maintain agreement about their overlapping effects, they may be freely mixed and employed toward a common goal. Maelstrom's algorithm is nothing more than a 'phenomenological sort' of phenomenologically modular scripts into a total order within which each can function properly.

Traditional ideas of software modularity express modules as being defined by interfaces with specified preconditions and postconditions on the use of each interface. For Maelstrom, instead modularity is based solely upon effect and postconditions; modularity is a requirement of the postconditions of the network after the script finishes. Every script must check its preconditions itself and only proceed if they are appropriate, so all scripts are expected to function correctly for all possible sets of preconditions.

Of course, what has been omitted so far in this discussion is the difficulty of writing scripts with the appropriate properties of environmental awareness, convergence, and homogeneity. In general it is difficult to construct such scripts while assuring script quality and reliability [10]. It is this difficulty that justifies the use of complex tools such as Cfengine [2, 3, 4] to fulfill that purpose instead.

Assuring Convergence

The problems of constructing convergent scripts have already been discussed in [9, 10]. One approach to ensuring awareness and convergence is to create an intelligent interface for dealing with the environment [7]. This interface hides the details of convergence from the script writer by checking for each change before making it inside of subroutines that accomplish the changes. As long as one interacts only with the environment through the lens of these subroutines, the script is guaranteed to be convergent. Without this discipline, convergent scripts are more complex than their non-convergent counterparts. Checking every parameter before changing it leads to doubling of script size with a commensurate increase in the difficulty and cost of script maintainability.

Assuring Homogeneity

Even if we can solve these problems, the new problem of assuring that scripts agree globally on their effects (homogeneity) gives rise to even more script complexity. One part of enforcing homogeneity is easy. If two scripts configure or control distinct domains with no overlap in parameters, such as a router and a distinct switch, we say the scripts have *orthogonal domains of change*. Such scripts are automatically homogeneous once they are convergent.

If, however, two scripts affect the same environmental parameters, they must somehow agree upon the appropriate values (or rules) for those parameters or homogeneity will not be possible. Scripts that overlap in function should ideally gather their configuration details from the same source. In this way, the only requirement for the scripts themselves is that they be convergent, and homogeneity is guaranteed by convergence. Again, this calls for a common interface; a library that accesses the same database of desired traits of a network for all scripts.

With both of these ingredients – convergent interface libraries and globally consistent access to configuration information – *homogeneity is automatically guaranteed*. But almost none of the modern solutions to scripting have the second property of consistent access to configuration information.

A beginning for this kind of globally consistent access to parameters may be found in the configuration files and common configuration interface of the Arusha Project [17, 18]. Even with powerful mechanisms such as Arusha's XML-based parameter inheritance, assuring both of these ingredients also requires substantial discipline on the part of the programmer; *all* access to the network or to configuration information must be through this mechanism. Any sloppiness in code – such as embedding configuration parameters into code or using other methods for accessing the network or the parameters – will lead to non-compliant scripts.

The Myth of Causality

Can the Maelstrom approach be applied to the problem of inferring the 'causes' of network problems? Probably not. The question of what 'caused' a specific problem usually turns out to be ill-formed. This is mainly because the action that seemingly repaired a problem need not be directly related to the true cause of the problem.

Proposition 8: **Causality is an abstract ideal that is meaningless in practice.**

Foremost, the reason for this is that any claim that the Maelstrom process can make about causality is only true about the computing environment in which it was observed. This environment is constantly changing, and many changes are not observable, so no prior inference can be useful in predicting future behavior reliably. Theoretical 'cause/effect' relationships can be used to guide the troubleshooting process, but these cannot reliably be inferred from observations during that process.

The concept of causality is plagued by 'latent variables' that affect the function of a system without being recognized as having any effect. Most recently, months of carefully checking DHCP and router configurations for a DHCP problem ended when we observed a DHCP server answering an ARP request with an incorrect IP address. The 'latent variable' turned out to be a defective network driver on the DHCP server – something we had never considered. The idea that troubleshooting is often a search for 'latent variables' justifies the 'phenomenological modularity' that allows one to easily add tests to a decision tree.

Eells [12] points out that the latent variable problem is much more severe than this simple case study illustrates. He shows that by an appropriate ignorance of latent variables one can 'prove' that smoking *prevents* cancer. In his example, one latent variable is the location of the test subjects, either in large cities or on rural farms.

While network debugging by lower-level staff can be represented by decision trees, persistent problems referred to the highest level of triage almost always involve latent variables that are not part of the decision tree at all. Thus it becomes very important to be able to add tests to the debugging process as latent variables are discovered without affecting the global integrity of the debugging process. This need justifies use of 'phenomenological modularity' to assure interoperability between scripts, and the complete intractability of determining precedences between the various tests justifies the execution time that it takes to infer the precedences automatically.

The idea of causality cannot be even exhibited within scripts that are appropriately homogeneous and convergent. In fact:

Proposition 9: **Script convergence obscures causal relationships.**

Causal relationships take the form of scripts (or actions) A, B, and C, where C always succeeds after A and always fails after B. In this case, A and B cannot be homogeneously convergent with one another, because they sufficiently diverge in what they do to either assure or break C. To add to this quandry, it is seldom true that a script enables or disables another in *all possible conditions*. This is also a mathematical idea that never appears in practice.

Limitations

When scripts are convergent, this does not imply that they are non-disruptive. Convergence is a post-condition describing the *resulting* state of the network *after* the script completes. A convergent script can still make *arbitrary* changes *while it executes*, up to rebooting servers or even re-routing services to new

servers. In this sense our definition is less stringent that that of Cfengine [2, 3, 4], which requires that a script avoid creating non-conforming equipment states while executing (which prevents it from rebooting servers).

This fact is the root of several limitations of the method. First, the method is inherently *serial*, because convergence and homogeneity are not otherwise guaranteed. *Maelstrom*, unlike make, cannot run scripts in parallel without presuming complete orthogonality of script domains of change. Future versions of *Maelstrom* may allow one to declare scripts as independent, to allow parallelism.

Perhaps the most severe hidden limit is the subtlety with which scripts must be developed in order to exhibit true and guaranteed homogeneity. The only reliable ways to assure this in practice are to use only scripts that administer non-overlapping domains and cannot behave inhomogeneously, or to force all scripts with overlapping domains to retrieve configuration information from a common and shared source.

Conclusions

Maelstrom is a relatively simple tool. But it is a result of complex and perhaps controversial changes in our thinking about the troubleshooting process. What we want is not necessarily what we need. It would be nice if we could predefine the script that accomplishes troubleshooting, by implementing a decision tree as a script. This does not work. It would be nice if we could predefine precedences between troubleshooting tasks. We cannot. It would be nice if the result of scripting was a description of the problem that was solved. As we often do not know the causes of problems when solving them as human troubleshooters, this is an idle dream.

The lesson of *Maelstrom* is that there are things we can do to automate troubleshooting without running into these roadblocks. We can employ convergence as a *replacement* for causal theory. We can concentrate on effects, and base the modularity of our automation upon consistency and agreement upon the effects we wish to achieve, and reliable reporting of effects, rather than agreement upon software interfaces or platforms. We can compensate for lack of agreement by sequencing.

But taking these steps also requires casting aside some commonly held values. We must remember that machine labor is cheap while staff labor is expensive, so that an inefficient machine process is sometimes preferable to an efficient one performed by a human. In utilizing machine labor, however, it is important to keep the automated process understandable and repeatable by a human, so that it can be verified, validated, and improved. *Maelstrom* does not use the most efficient strategy, but the most understandable one.

Research is itself a convergent process, so it is not surprising that many others have faced the same problems in other areas and come to some of the same conclusions. The need to limit ourselves to cases that are observable is echoed in current work on testing of distributed software systems [6, 11]. *Safety* (freedom from undesirable states) and *liveness* (freedom from deadlocks) are goals of both software engineering and system administration. Safety and liveness are difficult to assure, even when one has a complete model of what proper operation should be – something a network administrator often lacks.

It is often claimed that learning to be a painter is not so much learning to paint as it is learning to *see*. Here we have the same effect; we need to learn to trust our senses rather than our theories, our results more than our models. Only then can we sort out the maelstrom that is troubleshooting, and can truly know that we can "paint what we see."

Availability

Maelstrom is freely available from http::://www.eecs.tufts.edu/~couch/maelstrom. It is written in Perl-5 to be portable to almost every UNIX system in creation.

Acknowledgments

A work such as this is not the work of any individuals, but the product of an intellectual community of thinkers all willing to look beyond the obvious. We are indebted to every member of that community who stopped by for awhile to talk about the issues. Scott Pustay and John Hart were both willing ears as we initially developed these ideas. When things started taking shape, Michael Gilfix stepped in with many good ideas, applications, and even source code; the current neatness of *Maelstrom* is his doing. Our tireless system administrators Andy Davidoff and Shawn Doughty contributed both thought and problems we could try to solve. Ambrose Kofi Laing provided valuable references on combinatorics. David Krumme helped with last-minute checking of typesetting and technical results. Mark Burgess, Frode Eika Sandnes, and Adam Moskowitz all provided valuable feedback on the paper and its ideas. Max Ben-Aaron provided valuable suggestions for copy editing and prose. All of these people, not just ourselves, created and nurtured this idea – and we all draw strength from the community we have formed, together.

Author Biographies

Alva L. Couch was born in Winston-Salem, North Carolina where he attended the North Carolina School of the Arts as a high school major in bassoon and contrabassoon performance. He received an S.B. in Architecture from M. I. T. in 1978, after which he worked for four years as a systems analyst and administrator at Harvard Medical School. Returning to school, he received an M.S. in Mathematics from Tufts in 1987, and a Ph.D. in Mathematics from Tufts in 1988. He became a member of the faculty of Tufts

Department of Computer Science in the fall of 1988, and is currently an Associate Professor of Electrical Engineering and Computer Science at Tufts. Prof. Couch is the author of several software systems for visualization and system administration, including Seecube (1987), Seeplex (1990), Slink (1996) Distr (1997), and Babble (2000). In 1996 he also received the Leibner Award for excellence in teaching and advising from Tufts. He has assisted in maintaining the Tufts computer systems for Computer Science teaching and research since 1985, when he was a Ph.D. student. He can be reached by surface mail at the Department of Electrical Engineering and Computer Science, 161 College Avenue, Tufts University, Medford, MA 02155. He can be reached via electronic mail as couch@eecs.tufts.edu. His work phone is (617)627-3674.

A senior while this work was in progress, Noah Daniels received his BS in Computer Science, cum laude, from Tufts University in May 2001, and formerly studied Physics at Swarthmore College. He is currently a systems engineer for Analog Devices, Inc. in Wilmington, Massachusetts. His current position entails supporting a large network of SunOS and Solaris-based microchip testers in the backend manufacturing division of Analog. Prior to this, Noah was a system and networking consultant for Interliant, Inc. (formerly Net Daemons, Associates) in Woburn, Massachusetts. His interests within the realm of computer science include algorithms, system administration, and the Darwin kernel. When not in front of a keyboard, Noah enjoys playing the violin, running, and being around (and attempting to ride) horses. Noah can be reached via email at ndaniels@eecs.tufts.edu, and via postal mail at 509 Main St. Apt. B, Waltham, MA 02452.

References

[1] Adleman, L., "Short Permutation Strings," *Discrete Mathematics* **10**, pp. 197-200, 1974.

[2] Burgess, M., "A Site Configuration Engine," *Computing Systems* **8**, 1995.

[3] Burgess, M., and R. Ralston, "Distributed Resource Administration Using Cfengine," *Software: practice and experience* **27**, 1997.

[4] Burgess, M., "Computer Immunology," *Proc. LISA-XII*, 1998.

[5] Cai, M., "A New Bound on the Length of the Shortest String Containing all r-Permutations," *Discrete Mathematics* **39**, 1982, pp. 329-330.

[6] Cheung, S. C., and J. Kramer, "Checking Subsystem Safety Properties in Compositional Reachability Analysis," *Proceedings of the International Conference on Software Engineering*, Berlin, Germany, March 25-29, 1996.

[7] Couch, A., "SLINK: Simple, Effective Filesystem Maintenance Abstractions for Community-Based Administration," *Proc. Lisa-X*, Usenix Assoc., 1996.

[8] Couch, A., "Chaos Out of Order: A Simple, Scalable File Distribution Facility for 'Intentionally Heterogeneous' Networks," *Proc. LISA-XI*, Usenix Assoc., 1997.

[9] Couch, A., and M. Gilfix, "It's Elementary, Dear Watson: Applying Logic Programming to Convergent System Management Processes," *Proc. Lisa-XIII*, Usenix Assoc., 1999.

[10] Alva L. Couch, "An Expectant Chat About Script Maturity," *Proc. LISA-XIV*, Usenix Assoc., 2000.

[11] Duri, S., U. Buy, R. Devarapalli, and S. M. Shatz, "Application and Experimental Evaluation of State Space Reduction Methods for Deadlock Analysis in Ada," *ACM Transactions on Software Engineering and Methodology*, Vol. 3, No. 4, ACM Press, 1994.

[12] Eells, Ellery, *Probabilistic Causality*, Cambridge University Press, 1991.

[13] Galbiati, G., and F. P. Preparata, "On Permutation-Embedding Sequences" *SIAM Journal on Applied Mathematics*, Vol. 30, No. 3, pp. 421-423, May, 1976.

[14] Gilfix, M., and A. Couch, "Peep (The Network Auralizer): Monitoring Your Network with Sound," *Proc. LISA-XIV*, Usenix Assoc., 2000.

[15] Goldfarb, C., and P. Prescod, *The XML Handbook, Second Edition*, Prentice-Hall, Inc., 2000.

[16] Hansen, S. and T. Atkins, "Centralized System Monitoring with Swatch," *Proc. LISA VII*, Usenix Assoc., 1993.

[17] Holgate, Matt, and Will Partain, "The Arusha Project: A Framework for Collaborative Unix System Administration," *Proc. LISA XV*, USENIX Assoc., San Diego CA, 2001.

[18] Holgate, Matt, Will Partain, et al., "The Arusha Project Web Site," http://ark.sourceforge.net.

[19] Kleitman, D. J., and D. J. Kwiatkowski, "A Lower Bound on the Length of a Sequence Containing All Permutations as Subsequences," *Journal of Combinatorial Theory (A)* **21**, pp. 129-136, 1976.

[20] Libes, D., *Exploring Expect*, O'Reilly and Assoc., 1994.

[21] McCabe, T., "A Software Complexity Measure," *IEEE Trans. Software Engineering 2*, 1976.

[22] Mohanty, S. P., "Shortest String Containing All Permutations," *Discrete Mathematics*, Vol. 31, pp. 91-95, 1980.

[23] Oram, A., and S. Talbot, *Managing Projects with Make, Second Edition*, O'Reilly and Associates, 1991.

[24] Osterlund, R., "PIKT: Problem Informant/Killer Tool," *Proc. LISA-XIV*, Usenix Assoc., 2000.

[25] Sandnes, Frode Eika, "Scheduling Partially Ordered events in a Randomised Framework – Empirical Results and Implications for

Automatic Configuration Management," *Proc. LISA XV*, USENIX Assoc., San Diego CA, 2001.

[26] Savage, C., "Short Strings Containing All k-Element Permutations," *Discrete Mathematics*, Vol. 42, pp. 281-285, 1982.

[27] Schaffer, A. A., "Shortest Prefix Strings Containing All Subset Permutations," *Discrete Mathematics,* Vol. 64, pp. 239-252, 1987,

[28] Trocki, J., "Mon, the Server Monitoring Daemon," http://www.kernel.org/software/mon .

Performance Evaluation of Linux Virtual Server

Patrick O'Rourke and Mike Keefe – Mission Critical Linux, Inc.

ABSTRACT

Linux Virtual Server (LVS) is an open source technology which can be used to construct a scalable and highly available server using a collection of real servers. LVS load balances a variety of network services among multiple machines by manipulating packets as they are processed by the Linux TCP/IP stack. One of the most common roles for LVS is to act as a front end to a farm of web servers.

This paper documents a series of experiments performed on LVS by Mission Critical Linux, Inc. in a cooperative effort with Intel Corporation. The objective of these experiments was to evaluate LVS's ability to distribute web requests among several servers. We investigated a variety of LVS configurations and offer a comparison of LVS's ability to scale on Linux 2.2 versus Linux 2.4. In contrast to similar evaluations, our entire test effort was accomplished using open source software on Linux based platforms.

Our results show that in a uni-processor environment the performance of LVS on Linux 2.4 is on par with Linux 2.2, however in a multi-processor configuration, Linux 2.4 significantly surpasses Linux 2.2. LVS on Linux 2.2 actually exhibits minimal scaling in a multi-processor environment. We reveal the detrimental impact that multiple devices sharing interrupts can have on LVS throughput. A comparison of LVS to a commercial load balancer indicates that LVS is a viable alternative to the more expensive, proprietary solution. Our results show that LVS is nearly twice as cost effective in terms of price/performance when compared to the hardware based load balancer. Lastly, we document the steps necessary to enhance the capabilities of our load generator which in turn reduces the amount of client hardware needed.

Introduction

Linux Virtual Server [1] is an enhancement to the Linux operating system which permits a set of servers to offer network services (e.g., ftp, http, ...) as a single *virtual* server. The aggregate capabilities of the individual hosts, or *real servers*,[1] comprising the LVS cluster often exceed those of any one of the constituent systems as a stand alone entity. Reliability is also increased because services are backed by multiple machines, thereby avoiding the single point of failure one would have with a single server.

Although LVS benefits from a very active user community, there seems to be a void in the reporting of concrete performance data. This paper documents a series of experiments Mission Critical Linux, Inc. performed on LVS in order obtain a basic understanding of LVS's performance and scalability. One of the most common roles for LVS is to load balance HTTP requests [2], thus our primary focus was on the ability of LVS to distribute a prescribed workload among a set of backend web servers. A secondary objective was to get a sense of how well LVS compared to some of the commercially available, hardware based load balancers.

Linux Virtual Server Overview

This section is a brief overview of the LVS architecture and provides a background for the ensuing discussion. Please refer to Zhang [1] for a more complete and thorough discussion of LVS's internals.

An LVS cluster is made up of a *director* and one or more *real servers*. The LVS director is a modified Linux system whose responsibility is to distribute client requests among the real servers in a cluster. The real servers do the actual work of satisfying the request and generating a response back to the client. The director maintains a record of the number of requests being handled by each server and uses this information when deciding which server should receive the next request. An LVS cluster may also have a backup director which will take over in the event the primary director fails, however for the purposes of this report, we did not consider LVS's failover capabilities.

The real server can run any operating system and application which supports TCP/IP and ethernet.[2] Additional restrictions may be placed on the real servers depending upon the LVS configuration chosen.

LVS Configuration Methods

LVS employs several techniques for distributing IP packets among nodes in the cluster. One method uses network address translation (LVS-NAT) in which the headers of the packets are overwritten by the director. The director masquerades as the real server(s) and this creates the illusion that the real servers are being

[1] Real servers is the term used in the LVS documentation to denote the web server systems responsible for fulfilling the actual request.

[2] It may be theoretically possible to use an alternative to ethernet (e.g., an ATM network), however there may be issues in getting such an LVS configuration working.

contacted directly by the clients. The director *must* be the real servers' default gateway for an LVS-NAT configuration to work properly. As a result, every response from a server is returned to the client via the director. Although this scheme is sub-optimal from a performance perspective, it does have the benefit that the real server can run *any* operating system that supports TCP/IP.

A second LVS configuration uses direct routing (LVS-DR). As the name implies, each real server has an independent route back to the clients (i.e., internet) which is separate from the director. LVS-DR offers a significant performance advantage over LVS-NAT, but with some added complexity in configuring the cluster. Each node in the cluster is assigned the IP address of the virtual server (aka the VIP), but *only* the director is permitted to respond to address resolution protocol (ARP) requests (as a result all packets originating from the clients will be initially processed by the director). LVS-DR requires that the operating system on all real servers support this non-ARPing network interface in addition to TCP/IP.

The third and final method uses IP encapsulation, or IP tunneling (LVS-TUN), in which a packet intended for the virtual server is enclosed in another packet and retargeted to one of the real servers. As with LVS-DR, the responses from the servers do not need to return via the director and so LVS-TUN offers performance and scaling similar to that of LVS-DR. Some operating systems may not support IP tunneling, therefore an LVS-TUN configuration restricts the realservers to running one that does.

Tunneling allows the LVS administrator to put servers on separate network segments, whereas this is not possible with a direct routing configuration. We used a Gb Ethernet as our server network to avoid network congestion issues and consequently our experiments only concentrated on LVS-NAT and LVS-DR based configurations. Figure 1 provides a comparison of the differences between the various LVS configurations.

Scheduling Client Requests

Client requests are distributed among the real servers based on IP address, protocol (i.e., TCP, UDP) and port number; commonly referred to as *Layer 4* switching. LVS supports several different scheduling algorithms that are settable during the configuration of the LVS cluster. The scheduler is responsible for determining which real server will receive the next client request. The simplest scheduler is round robin which simply circulates requests among each real server in a round robin fashion. All the LVS configurations in our testing used the round robin scheduler since every real server had similar attributes (e.g., CPU speed, memory, network connectivity). Refer to LVS documentation [4] for a complete and thorough discussion of all the LVS scheduling algorithms available.

Test Environment

One of the most challenging aspects of this project was devising a hardware and software configuration capable of stressing the director. The hardware vendors and trade magazines that have reported results to date appear to use either closed tools or hardware

Implementation	*Advantages*
LVS-NAT	Any OS with TCP/IP support Servers can use private IP addresses Only director needs public IP address
Direct Routing	Better scalability than LVS-NAT Director only handles client-to-server half of connection Response packets follow separate routes to clients Doesn't have IP tunneling overhead
IP Tunneling	Scales similar to LVS-DR Director schedules requests to the different servers Servers return directly to clients Servers can be on different network than director
Implementation	*Disadvantages*
LVS-NAT	Does not scale as well as LVS-DR or LVS-TUN Director becomes a bottleneck; because packets must pass through it in *both* directions
Direct Routing	director and servers must be on same network segment Servers need public IP addresses Server OS requires non-ARPing network interface
IP Tunneling	Server OS must support IP tunneling Servers need public IP addresses Overhead of IP encapsulation

Figure 1: Comparison of LVS features (see [3]).

based solutions (e.g., such as SmartBits) for generating a client workload [5] [6]. The only multi-client Linux based tool we came across was a tool called *Tarantula* developed by Arrowpoint Communications [7], but unfortunately this tool is not in public domain. We wanted a rather simple client workload so we could focus our attention on the LVS director, as opposed to tuning the web server software, therefore we avoided a complex workload generator such as SpecWeb.

Hardware Configuration

LVS Director

The hardware selected as the LVS director was chosen because we felt it was representative of a "typical" server based system on the market today. The system was configured with 512 MB SDRAM, 1-4 500 MHz Intel Pentium III Xeon Processors (512 KB L2 cache) and the ServerWorks* ServerSet II HE chipset. The director was equipped with two Intel PRO/1000 F Gb ethernet cards, each connected to a single Cisco Catalyst 4006 switch, which also acted as a gateway between the server and client subnets (see Figure. 2).

Real Servers

There were between one and four real servers, three of them contained four 500 MHz Pentium III CPUs and one had four 550 MHz Pentium III CPUs using the SC450NX motherboard/chipset. Each system was booted with one gigabyte of RAM and an Intel PRO/1000 F Gb Ethernet card connected to the Cisco Catalyst 4006 switch.

Client

There were two sets of clients. The first set consisted of four systems and was connected to the Catalyst 4006 using Gb Ethernet (two via Intel PRO/1000 F and two). Two of these contained eight 550 MHz Intel Pentium III Xeon Processors with Intel's Profusion chip set and one gigabyte of memory. The other two were dual 500 MHz Intel Pentium III Xeons, one gigabyte memory with the L440GX motherboard/chip set.

The second set of clients was ten 233 MHz Pentium II CPUs, 128 MB of RAM connected to the Catalyst 4006 via 100 Mb eepro100 cards.

Network Infrastructure

The test environment consisted of two network segments with the Cisco Catalyst switch acting as the gateway between them. The server network was a Gb Ethernet and each server was directly connected to a Gb port on the Cisco switch. The clients with Gb cards were also connected to Gb ports on the 4006 switch, while the remaining ten Pentium II clients were connected into 100 Mb ports on the Catalyst.

Software Configuration

We installed Redhat's 6.2 distribution of Linux (which is based on version 2.2.14 of the Linux kernel) on each of the systems. Some initial network [8] testing (Figure 3) indicated we could achieve better throughput using a 2.4 based kernel.

Figure 3 shows the results of a connectivity test between one of the eight way clients and the LVS director node. These tests were run with each machine connected via the Cisco switch with an MTU of 1500 bytes, as opposed to a back to back test (in which the two hosts are connected directly two each other). The plots show the network throughput as packet size is increased from one byte to a maximum of twelve megabytes. The "2.2" curve reports the results with each system running Linux 2.2.14, while the plot

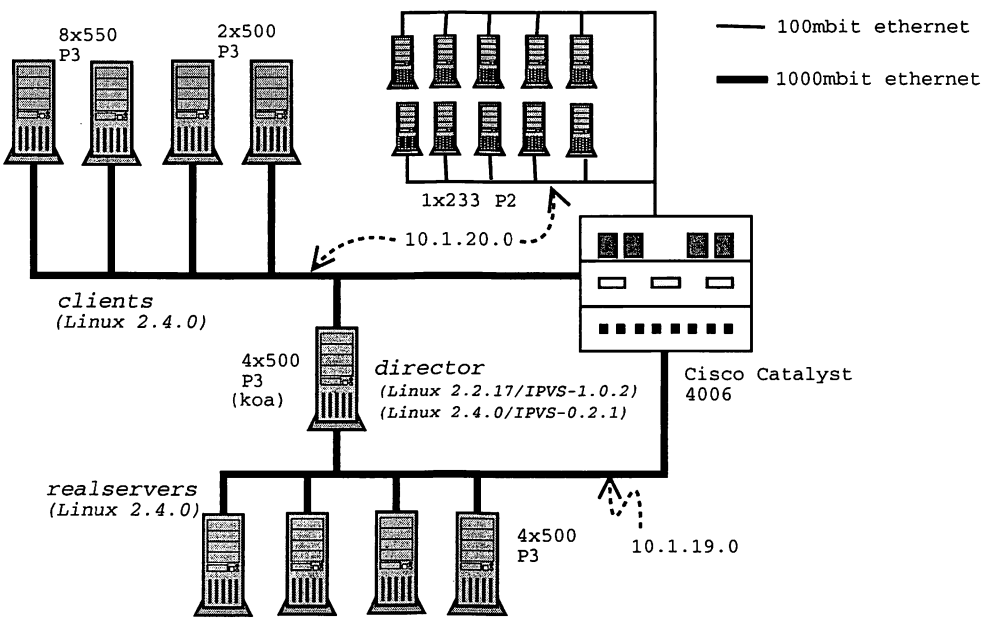


Figure 2: LVS test hardware.

labeled "2.4" is with each machine running Linux 2.4.0. The PRO/1000 cards were in 64 bit/66 MHz PCI slots on each system for these tests. The "100 Mb netpc" shows the Pentium II based clients are capable of saturating their 100 Mb link to the Cisco switch.

As a result of these connectivity tests we installed version 2.4.0 of the Linux kernel on each client and real server. This also required us to upgrade to revision 2.4.0 of the *modutils* package. A minor patch was also required in order to get the the e1000 device driver to load on 2.4.0.

Figure 4 summarizes the versions of the major software components installed on the test hardware.

Director Software

Although Redhat bundles a version of LVS with its distributions, we preferred to use a later version of the LVS software as well as a newer kernel. We used the latest 2.2 kernel available at the time we began our testing which was Linux 2.2.17 with version 1.0.2 of LVS. For testing a 2.4 based director we used a 2.4.0 kernel with LVS version 0.2.1. The LVS software was linked directly into the kernel for both 2.2 and 2.4 based directors.

Client Workload Generator

A key element to testing web based systems is the software used to generate client requests. We wanted the software to be lightweight so that a high number of requests could be initiated using a minimum number of clients. We also preferred to use an existing tool, rather than create our own so we could report the results of a "known entity," and last but not least an open source tool was desirable.

Julian Anastasov created a very efficient tool for stressing an LVS director called *testlvs* [9]. Testlvs is capable of simulating a large number of clients making connection requests by sending raw IP packets with spoofed source addresses. The connections are never fully established and as a result valid web requests are never actually made of the real server(s). We found it difficult to report a metric using testlvs, therefore we opted not to take advantage of it.

Component	*Version*
Linux distribution	Redhat 6.2
2.2 based kernel	Linux 2.2.17
2.2 based LVS	ipvs-1.0.2
2.4 based kernel	Linux 2.4.0
2.4 based LVS	ipvs-0.2.1
C compiler	egcs 2.91.66
Linker	ld 2.9.5
C library	libc-2.1.3
Modutils	modutils-2.4.0
PRO/1000 driver	e1000-2.5.14
Web software	apache-1.3.12
Client load generator	httperf-0.8

Figure 4: Software components.

David Mosberger and Tai Jin created a tool called *httperf* [10] which is designed to measure web server performance. Httperf issues connection requests to a web server at a specified rate and measures the actual reply rate along with various other metrics (e.g., average response time, percent of connections in error, network throughout, ...). On a server capable of sustaining the offered connection rate, the number of replies will equal the number of requests and these replies will be received within the specified time-out

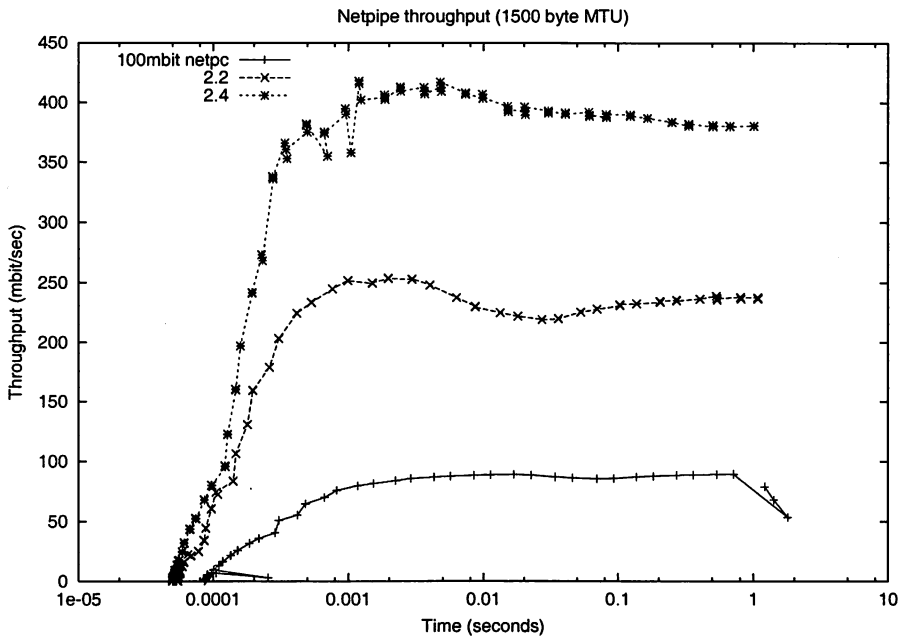


Figure 3: Netpipe signature graph.

period. Httperf provides many options to control its behavior, but it has no facility for being executed on multiple clients which made running it on fourteen clients simultaneously cumbersome. Therefore we created a small utility which would execute httperf on each client, wait for it to finish and collect the output into one central log.

Testing Methodology

The objective of this section is to offer an accurate and clear understanding of how these experiments were conducted. Our primary goal was to stress the LVS software (i.e., the director node) and so we attempted to keep our workload as simple as possible. As a result we explicitly avoided some parameters which are otherwise very important to the overall performance of a web farm. The clients were restricted to requests based on the HTTP 1.0 protocol. Each request was for the same web page which ensured that the web data would remain in the server's file cache, thereby eliminating any idiosyncrasies of file I/O on the web servers. The requested page was small enough (628 bytes) to be transmitted in a single packet without the need for fragmentation (given an MTU of 1500 bytes).

We recognize that a more comprehensive test suite must address issues like HTTP 1.1, requests for variable sized data and complexities such as dynamic content. Introducing all of these features would have made it much more difficult to evaluate LVS's ability to load balance web requests due to the additional variables in the mix. This is also the reason why a comprehensive web benchmark like SpecWeb was not used to generate the work load.

Apache Tuning

The Apache configuration file on each real server was altered to disable logging and to keep a sufficient number of httpd daemons available so as to minimize overhead in responding to client requests. Below are the Apache tunables altered:

```
#CustomLog /var/log/httpd/access_log \
     common # turn off logging
LogLevel crit
MinSpareServers 200
MaxSpareServers 200
MaxClients 1500
MaxRequestsPerChild 0
     # servers never go away
```

Linux Tuning

The amount of socket memory on each system (client, director and real server) was increased to five megabytes to allow for a large TCP window size:

```
echo 5242880 > /proc/sys/net/core/rmem_max
echo 5242880 > /proc/sys/net/core/rmem_default
echo 5242880 > /proc/sys/net/core/wmem_default
echo 5242880 > /proc/sys/net/core/wmem_max
```

In order to maximize the number of concurrent connections a given client could create it was necessary to increase the number of per-process file descriptors as well as system wide limit on files and local port numbers:

```
ulimit -n 100000
sysctl -w 'fs.file-max=150000'
sysctl -w 'fs.inode-max=32768'
sysctl -w 'net.ipv4.ip_local_port_range=1024
                                        40000'
```

LVS Configuration

All web servers were essentially the same type of hardware and so we always configured them as being equal in weight. The clients all requested the same web content and we did not experiment with the various LVS scheduling algorithms; we simply stayed with round robin. We avoided extensive testing on a 2.2 based director in an SMP configuration because earlier network testing done at Mission Critical Linux found additional processors did not improve network scalability on Linux 2.2. We also did not test a 2.4 LVS-NAT configuration because we had difficulty getting the director to successfully masquerade for the real servers, which turned out to be a known limitation of LVS in 2.4. A workaround for LVS-NAT has since been provided in ipvs-0.2.2, but unfortunately time did not permit us to re-run our tests.

httperf

Httperf was run on each client machine such that the desired connection rate was distributed evenly among all fourteen clients. Each run lasted three minutes as suggested by [10] so the server farm could reach steady state. The client time out specified was one second, thus if a connection was not established within that period it would be flagged as an error. The offered connection rate was increased on each client until we reached the maximum amount of connections any given system was capable of sustaining. This resulted in the largest aggregate connection rate our test bed could sustain as 14140 connections per second, or approximately 1010 connections per second, per client. This seemingly artificial barrier is the result of a limitation of the __FD_SETSIZE macro in /usr/include/bits/types.h which restricts the number of files (i.e., connections) to 1024.[3] Fortunately a connection rate of 14140 connections per second was sufficient to saturate all tested LVS configurations with the exception of the four processor, four real server 2.4 LVS-DR configuration.

The aggregate reply rate was calculated by summing the individual reply rates reported by each client. Each data point was repeated three times to account for any aberrations and the results reported below represent the mean of these three runs. It was also necessary to delay 120 seconds between each iteration to allow for any lingering connections to leave the TIME_WAIT state.

Results

The results in this section are based on a series of test runs as shown in Figure 5. Some additional workloads

[3]We were eventually able to overcome this limitation by modifying the httperf source code, but this was not used in our testing.

were done on specific configurations in order to gain further understanding of a particular result.

As a control we also ran the workload directly against a single web server without LVS. This served to provide baseline data so we could easily evaluate the impact of LVS. It should be noted a second control case would be with the director node forwarding packets without LVS, however we did not investigate this due to time constraints.

Linux 2.2

Figure 6 shows the behavior of a single processor director running 2.2.17 with one, two and four real servers respectively. The plot labeled "Direct" represents the reply rate sustained by a stand alone web server running the configuration. The direct and LVS-NAT curves both peak at around 4000 connections per second, but we see that with LVS-NAT the sustained workload is less. This is a result of the extra overhead associated with LVS-NAT, i.e., the fact that return packets from the real server must pass through the director. The reason additional real servers offer no benefit is because one alone is sufficient to saturate the director, therefore we cannot take advantage of the extra servers.

Figure 7 reports on the LVS-DR configuration running on a single CPU, 2.2.17 based director. LVS-DR with a single real server ("DR, 1 RS") appears to be on par with the direct case. The addition of a second real server ("DR, 2 RS") provides a jump from about 4000 connections per second to approximately 6700 connections per second. This increase can be explained by our observation that a single web server saturates at 4000 connections with the small request size chosen, but the additional server allows us to distribute the load among two servers. Hardly any improvement at all is seen when we increase the number of real servers from two to four ("DR, 4 RS").

Based on subsequent results seen with a 2.4 SMP based director, we suspect we are hitting a limit on the packet processing ability of the single CPU in the director.

When describing our test methodology we noted that we did not expect linear scaling as more processors were added to a 2.2 based director. To verify this we performed a test on the largest 2.2 configuration we had available (four CPUs, four real servers) and compared that to a single processor director with the

Director	*Config*			*Real servers*
	uniprocessor	dual processor	quad processor	
2.2.17 uni	NAT	X	X	X
2.2.17 uni	DR	X	X	X
2.4.0 uni	DR	X	X	X
2.4.0 dual	DR	X	X	X
2.4.0 quad	DR	X	X	X

Figure 5: LVS test matrix.

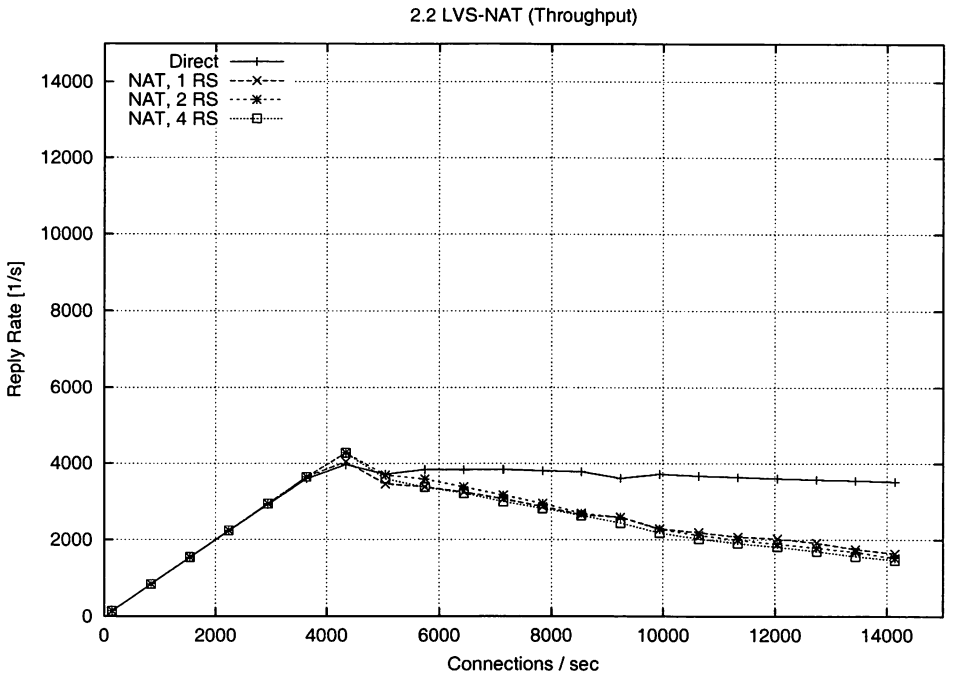


Figure 6: Single CPU Linux 2.2.17, LVS-NAT throughput.

same number of servers. Figure 8 shows the results of this test and we do see only a marginal improvement. The reason for such a small improvement is that much of the networking code in Linux 2.2 still executes under the auspices of the global kernel lock. For all intents and purposes the 2.2 TCP/IP code is really running on one CPU, however there still are moments when other useful work can be accomplished with the additional CPUs, therefore we do see a small improvement.

Finally Figure 9 combines all the 2.2 based runs so we can easily view a comparison of LVS-NAT versus LVS-DR capabilities. As expected we can sustain a significantly higher connection load with an LVS-DR based configuration than is possible with LVS-NAT.

Linux 2.4

Figure 10 shows the connection rate sustainable by a 2.4 based director using LVS-DR. As a baseline we again include the case of running directly against the web server without LVS ("Direct") and show the impact of adding one, two and four real servers; curves "DR, 1 RS," "DR, 2 RS," "DR, 3 RS" respectively. This graph is similar to that of the 2.2

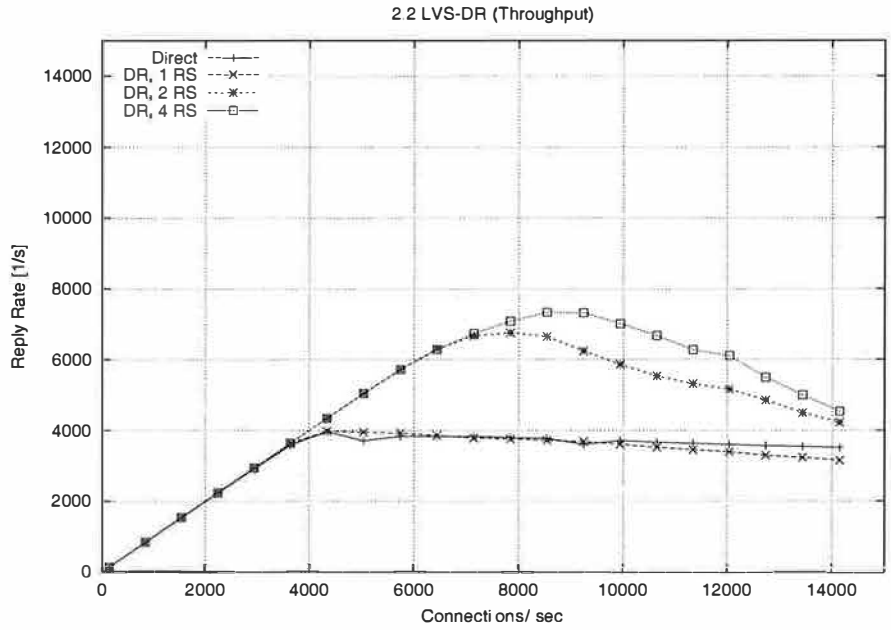


Figure 7: Single CPU Linux 2.2.17, LVS-DR throughput.

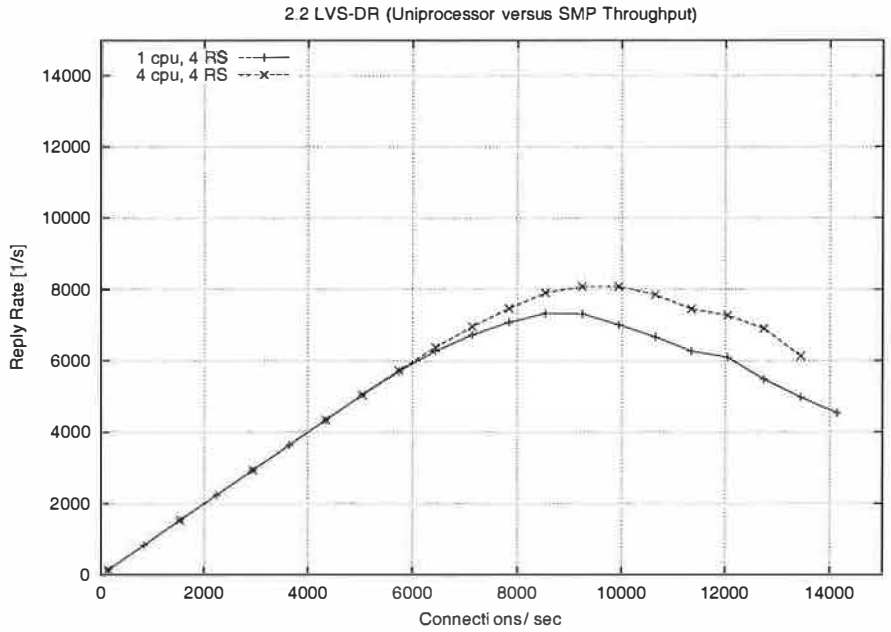


Figure 8: SMP Scaling on Linux 2.2.17.

LVS-DR seen in Figure 7 and we see just how close they are in Figure 11.

Figure 11 indicates that a single processor 2.4 based LVS-DR configuration is *very* close in performance to that of a 2.2 based director, in fact it shows 2.2 is slightly better than 2.4. Joseph Mack pointed out this slow down is because LVS in 2.4 is implemented via the hooks provided by the Netfilter subsystem, whereas in 2.2 LVS ran independently, i.e., the LVS patch was applied directly to the Linux IP networking code. According to the data we collected, one of our single web servers can sustain a connection rate of approximately 4000 connections per second. Therefore if one were to have perfect scaling, an LVS-DR configuration with four webservers should theoretically be able to support 16000 requests per second. While we do see a nearly linear jump from one to two servers (on both 2.2 and 2.4), it seems the additional two servers when going from two to four provides almost no gain at all. We believe this to be a limitation of the single processor in the director to handle the network traffic.

Figure 12 shows our ability to scale with two CPUs as we add real servers. Increasing from one real

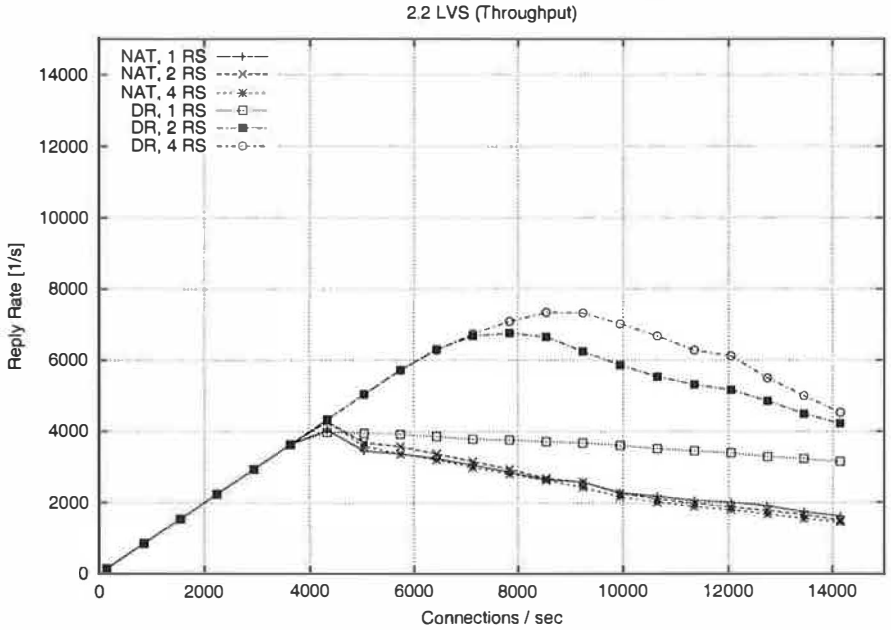


Figure 9: Single CPU Linux 2.2 LVS-NAT vs. LVS-DR scaling.

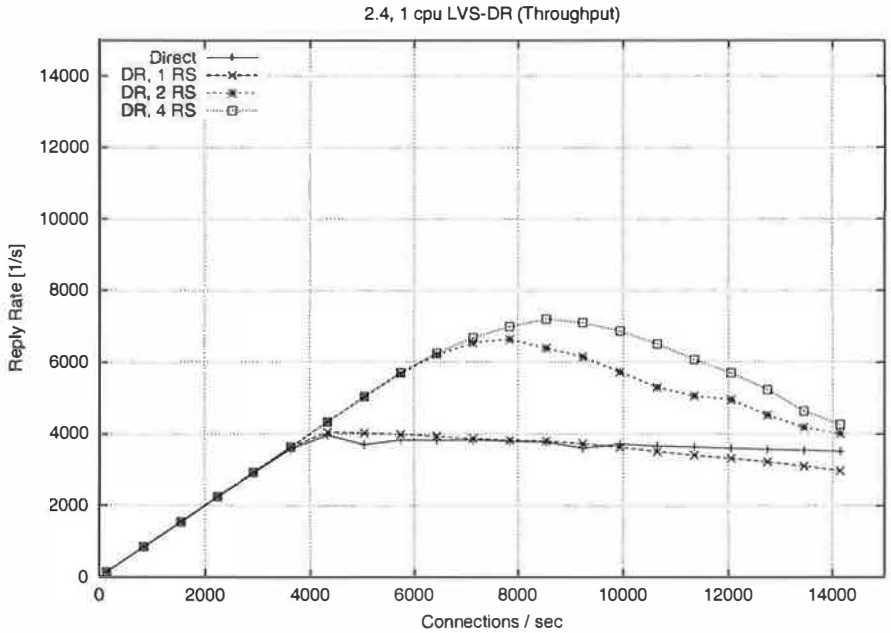


Figure 10: Single CPU, Linux 2.4.0, LVS-DR throughput.

server to two provides a nearly linear increase in overall connection rate, a jump from 4000 to around 7700 connections per second. According to our results, a two processor LVS-DR director with two servers *outperforms* a uni-processor with four servers. We suspect the reason is because the TCP/IP layer in 2.4 is fully multi-threaded, so the additional server's load can be handled in parallel by the second CPU. We are no longer contending for the global kernel lock as is the case in 2.2. The addition of two more servers brings us closer to theoretical limit of 16000 requests per second, but in this case we top out at around 10000 connections per second.[4] We suspect the reason being that the request rate has exceeded the ability of two processors to sustain.

We see in Figure 13 that more CPU power brings us even closer to our theoretical limit of 16000 connections per second. This is evidence for how well parallelized the TCP/IP stack is in Linux 2.4. In fact our set of clients in their current configuration is

[4]httperf was specified to make only one request per connection, so we use the terms connection and request interchangeably.

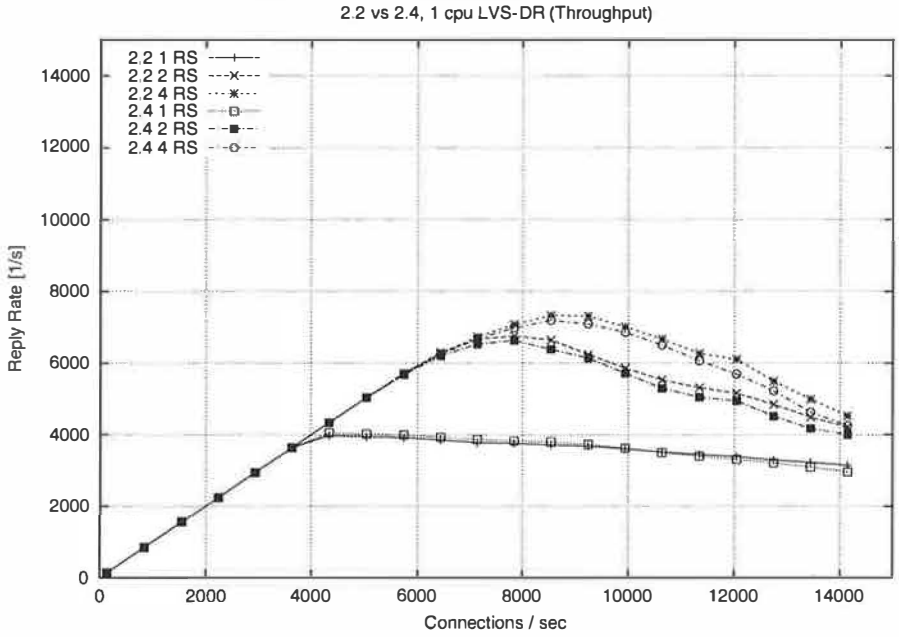


Figure 11: Single CPU, Linux 2.4 vs. Linux 2.2.

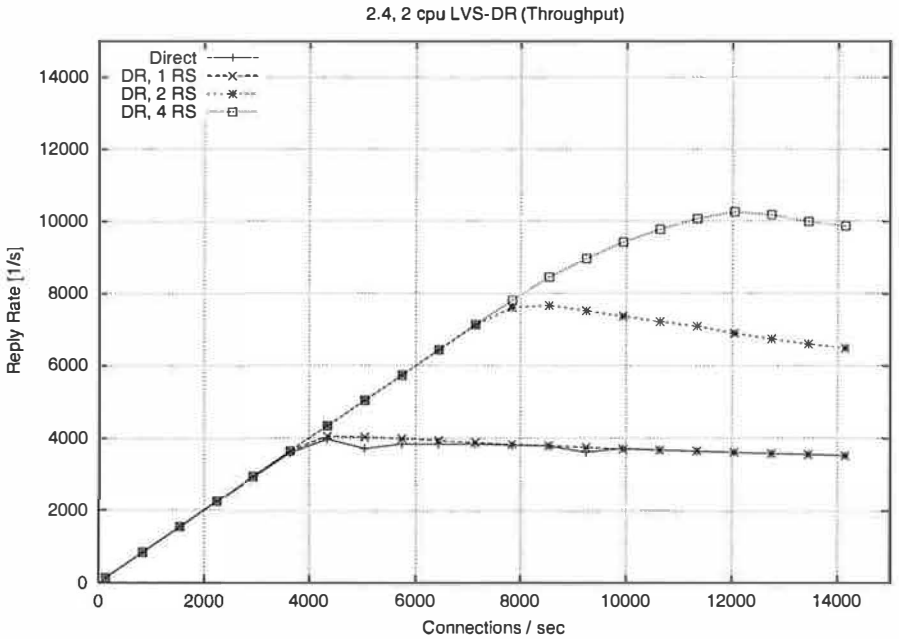


Figure 12: Linux 2.4, two CPU LVS-DR throughput.

capable of driving the LVS cluster to saturation. You can see from the shape of the curve "DR, 4 RS" in Figure 13 that it looks as if we will peak at around 13000 connections per second.

Figure 14 illustrates the ability of a 2.4 based LVS-DR to scale as we add processors with real servers held constant at four.

Director Load

Earlier results indicated the peak load that a single real server could sustain was approximately 4000 connections per second. The question remains however if we are limited due to network saturation or by the capabilities of the real servers. Httperf allows us to estimate how much network bandwidth our tests consumed because it reports the amount of network I/O each client received. Recall that the requested web content was 628 bytes in length, and according to httperf each request contained 288 bytes of packet header, so each reply received corresponds to 916 bytes of network data transmitted. We can therefore estimate the amount of network traffic by taking the sum of data received for each client during a particular test run.

Figure 15 shows the aggregate network I/O reported by httperf at the highest sustainable reply rate

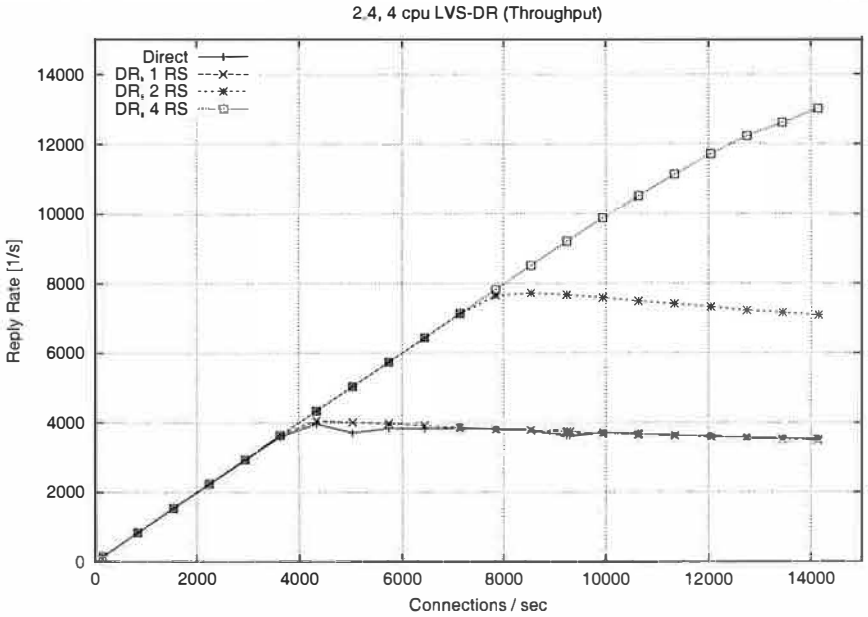


Figure 13: Linux 2.4, four CPU LVS-DR throughput.

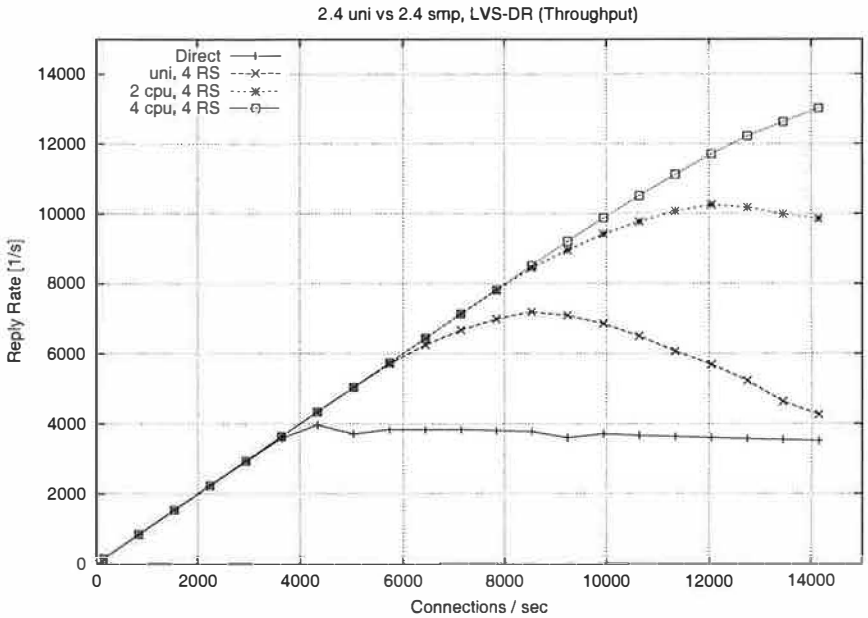


Figure 14: LVS Scaling on Linux 2.4.

for several selected test runs. We are not close to saturating the 100 Mb connections on the smallest clients, let alone the Gb interfaces on the director and real servers, even at our peak connection rate of 13019.3 replies per second.

We would also like a sense of the load on the cluster nodes themselves during peak activity. Therefore we sampled vmstat(8) output during runs of the highest sustainable connection rates for various configurations. Figure 16 reports the CPU utilization figures for a single CPU director when front ending one, two and four real servers respectively. At an offered connection rate of 4200 connections per second the single processor director is lightly loaded with an idle time of 85 percent. The single real server on the other hand is fairly loaded with less than 20 percent idle time. The load on the director increases with the number of real servers and we achieve an average system time of 55 percent with four real servers (peaks of 80 percent system time were seen for the four real server case).

Figure 17 shows the CPU utilization figures for a two CPU director. We did not include the numbers for the one real server scenario because they were very similar to the uniprocessor director shown previously. As with the single CPU director case, we are capable of saturating both the two and four real server configurations. The load on the two CPU director is higher than a single, but that coincides with an increase in the sustainable connection rate as well, a peak of 7115 replies per second. We did observe periods of greater than 90 percent system time on the two CPU director when running with four real servers.

In Figure 18 we show the utilization figures for the four CPU director with two and four real servers respectively. The added CPU cycles allow us to reach a reply rate over 12000 connections per second, but we still have a nearly 40 percent idle time on the director. At this connection rate we encounter the limitation in httperf which prevents us from reaching a higher load on the director.

Configuration	Con Rate	Reply Rate	Network I/O (Mb/sec)
Direct	4340	3962.93	3.8
2.2 NAT (1 CPU, 4 RS)	4340	4270.00	4.1
2.2 DR (4 CPU, 4 RS)	9240	8072.30	7.7
2.4 DR (4 CPU, 4 RS)	14140	13019.30	12.5

Figure 15: Aggregate network I/O.

Real Servers	Con Rate	Reply Rate	Node	Mean CPU utilization times (standard deviation)		
				User	System	Idle
1	4200	3897.9	director	0.00 (0.00)	11.04 (5.81)	85.04 (17.43)
			rs1	25.34 (13.24)	53.77 (27.62)	18.66 (38.71)
2	7000	6466.2	director	0.00 (0.00)	50.92 (18.41)	49.08 (18.41)
			rs1	29.91 (4.97)	68.26 (7.07)	1.82 (8.59)
			rs2	30.06 (4.19)	67.90 (9.18)	2.10 (11.67)
4	7000	6466.2	director	0.00 (0.00)	54.14 (20.30)	45.83 (20.33)
			rs1	15.40 (5.18)	82.86 (7.25)	1.74 (7.27)
			rs2	17.29 (5.60)	81.21 (6.34)	1.47 (6.05)
			rs3	15.61 (8.14)	83.36 (10.69)	0.97 (4.78)
			rs4	12.00 (5.48)	84.42 (14.82)	3.58 (16.09)

Figure 16: CPU utilization, single CPU, 2.4 LVS-DR.

Real Servers	Con Rate	Reply Rate	Node	Mean CPU utilization times (standard deviation)		
				User	System	Idle
2	7700	7102.0	director	0.00 (0.00)	47.84 (16.38)	52.11 (16.40)
			rs1	30.94 (3.41)	66.71 (6.88)	2.31 (9.63)
			rs2	33.35 (2.81)	65.35 (4.70)	1.32 (7.18)
4	7700	7115.1	director	0.00 (0.00)	63.08 (23.72)	36.92 (23.72)
			rs1	22.47 (4.24)	73.29 (12.62)	4.21 (16.35)
			rs2	25.30 (3.49)	71.97 (7.71)	2.77 (10.11)
			rs3	23.43 (3.85)	72.89 (10.74)	3.74 (12.73)
			rs4	20.70 (2.28)	77.52 (6.21)	1.73 (6.59)

Figure 17: CPU utilization, two CPU, 2.4 LVS-DR.

Sharing IRQs

Profiling the director via readprofile(1) showed we were spending a great deal of time in the AIC SCSI controller's interrupt service routine on both 2.2 and 2.4 based configurations. Further investigation revealed that the e1000 card and the SCSI controller were sharing the same IRQ which resulted in do_aic7xxx_isr() being called for *every* packet received. Linux 2.4 provides a config option to use the APIC controller available on IA32 based systems in a uni-processor environment (CONFIG_X86_UP_IOAPIC).[5] This resulted in allowing us to use different IRQs for the e1000 and AIC controller.

Figure 19 shows the dramatic benefit of eliminating the IRQ contention. With two real servers our sustainable peak increased from 6600 to 7400 requests per second. In the case of four servers the peak sustainable rate went from 7200 to 8300 connections per second.

We re-ordered the manner in which interrupt handlers register themselves, while leaving the IRQ shared between the e1000 and AIC. As a result the e1000 interrupt routine would be called prior to the AIC driver's. The hope was that the e1000 would handle the interrupt and the kernel would skip the unneeded call to the AIC driver. The device driver reordering had no effect on performance because the interface between handle_IRQ_event() and driver functions does not have a mechanism to signal that an interrupt has been processed. Thus the kernel continued to call each driver as before the re-ordering experiment. The APIC controller permitted us to place the AIC and e1000 drivers on separate IRQs, thereby eliminating the overhead of the AIC interrupt routine, and resulting in better performance.

[5]The APIC is used by default in SMP kernels.

LVS vs. A Hardware Load Balancer

Both hardware and software solutions are presently available for solving the load balancing, packet redirector problem inherent in the development of a scalable web farm. Linear scalability (or as close to it as possible) of a web farm is desirable since we want to easily predict the number of servers to add as the request load increases. Any bottlenecks will

Real Servers	Con Rate	Reply Rate	Node	Mean CPU utilization times (standard deviation)		
				User	System	Idle
2	7000	6478.0	director	0.00 (0.00)	27.11 (9.08)	72.89 (9.08)
			rs1	26.35 (5.89)	71.44 (8.92)	2.21 (9.94)
			rs2	32.28 (10.77)	65.97 (9.82)	1.69 (4.90)
4	14000	12296.5	director	0.00 (0.00)	61.08 (19.57)	38.86 (19.55)
			rs1	29.48 (1.92)	69.76 (3.17)	0.76 (3.50)
			rs2	31.81 (6.23)	65.13 (11.33)	3.03 (16.34)
			rs3	28.73 (5.34)	64.36 (13.19)	6.94 (17.68)
			rs4	25.91 (4.91)	70.24 (11.52)	3.76 (15.82)

Figure 18: CPU Utilization, four CPU, 2.4 LVS-DR.

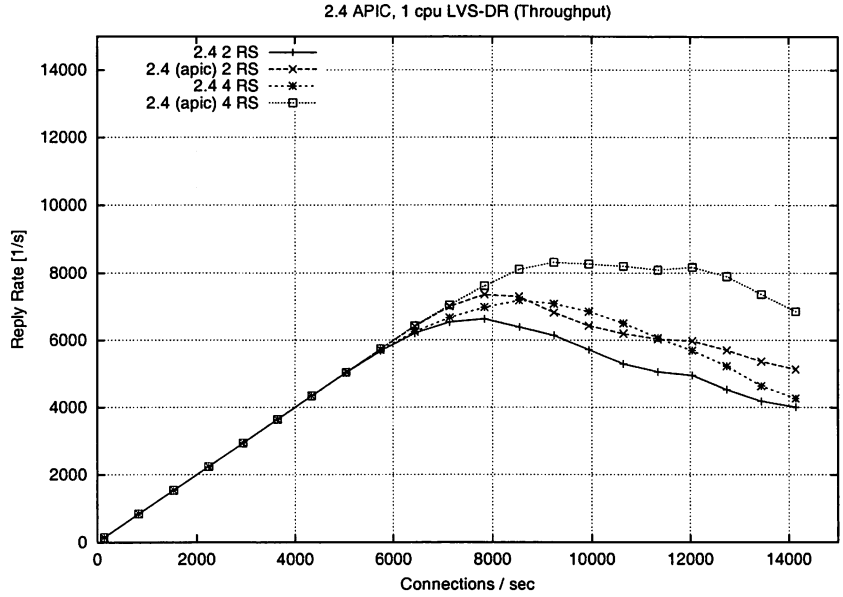


Figure 19: APIC support for uniprocessor Linux 2.4.0.

prevent the web farm from scaling predictably and in fact, may provide less processing power than was previously available under a lighter request load.

For a web server without bottlenecks and whose request load scales linearly, the following is true:[6]

```
connection rate =
      average request rate =
      average reply rate
```

Given this identity, we can try to compare our software-based results for LVS against those reported on hardware products such as ArrowPoint's CS-150 Content Smart Web switch [11]. The test methodology employed by ZD Labs on the CS-150 enables NAT and round-robin load balancing so we will use the LVS-NAT configuration with the highest throughput for our comparison; namely Linux 2.2 on a uni-processor director with four real servers and round-robin scheduling. For a .01 percent error reply rate (all of which are client time out errors), such a configuration produces the reply rate shown in Figure 20.

connection rate	3640 conn/sec
average request rate	3640 requests/sec
average reply rate	3639.77 replies/sec
average reply (response) time	2.11 ms

Figure 20: Reply rate for uniprocessor 2.2 LVS-NAT with 0.01 error rate.

Two hardware products we can compare the above LVS based solution against are ArrowPoint Communications' Content Smart Web switches; models CS-150 and CS-800. Each client in ZD Labs's tests repeatedly requested a 114 byte HTML web page, this should be comparable to the 628 byte web page requests used during our LVS testing. Joseph Mack [12] shows that network transmission times are approximately the same for sizes less than the MTU (1500 bytes). ZD Labs did not provide any information about errors that occurred during their testing, even though they gathered this information. This study reports the following results:

- CS-150: 12 servers, 15 clients: average load of 4400 GET requests/sec
- CS-800: 47 servers, 30 clients: average load of 16600 GET requests/sec

The CS-150 and CS-800 results are similar to those obtained for our NAT-based LVS test. Although LVS-NAT has raw performance that is on average 20 percent lower than the ArrowPoint CS-150, the ArrowPoint switch costs approximately $18,000[7] compared to an estimated value of $7,500 for the four processor director. The LVS solution is approximately twice the cost effectiveness of the CS-150 if one computes the cost per requests/second. The Arrowpoint comes out to $4.09 per request/second ($18,000/4400) whereas the LVS based configuration is $2.06 per request/second ($7500/3640). Also note that only *four* real servers were used by the LVS test whereas the CS-150 configuration used twelve. These results should be revisited once the 2.4 LVS-NAT functionality is stable and fully operational since it is likely that it will outperform ArrowPoint's CS-150 web switch.

[6] At least for a web server performance measuring tool such as httperf and presumably this holds for Tarantula – the tool used by ZD Labs in its tests

[7] As of April 17, 2000.

Future Work

A significant number of LVS features would benefit from more performance and scalability analysis in addition to the analysis we have done. It would be interesting to see how LVS behaves under a more varied workload instead of the static data we used in this report. This would enable one to study the effects and benefits of the various scheduling algorithms offered. Commercial web sites currently employ a variety of different protocols (e.g., HTTP 1.1, SSL, ...) and it may be interesting to see how well LVS can handle such workloads.

We had to skip testing a 2.4 based NAT configuration because the software is still evolving. Since that time LVS-NAT has become functional in 2.4 so it would be beneficial to compare it to the 2.2 version. LVS-NAT is an important configuration because it allows your real server to be any platform, such flexibility would be important to anyone wishing to product-ize an LVS based solution so as to avoid making restrictions on the types of platforms which can function as servers.

It would also be of great value if one could compare a server farm load balanced with LVS to one front ended by a commercial alternative such as Alteon or Foundry. Such a test would provide an users with an accurate price/performance comparison so they could make an informed decision on which technology is best for them. We found it difficult to ascertain the true capabilities of a commercial product from the vendor's literature, not to mention the obvious lack of objectivity. It seems as if each vendor reports performance results in a slightly different manner, or are somewhat vague in how they conducted the tests (e.g., number of clients, what type of operating system was used and so on).

There also is a lack of standardization on testing and evaluating load balancers in general. This is evidenced by the widely varying statistics reported from commercial load balancers. An industry-wide, standards based test suite would permit a direct comparison to be made between a software based approach such as LVS and a hardware solution.

Conclusion

Our experience with Linux Virtual Server has been a positive one overall. Although not an exact

comparison, we give an example in of how LVS can be a reasonable alternative to a more expensive hardware offering. The 2.2 LVS-NAT was unable to sustain as high a steady state as a stand alone web server, this warrants further investigation to determine if the performance of LVS-NAT can be improved. The Linux 2.4 based platform shows promise in its ability to scale as we add processors. We illustrated how important it is for the director to be properly configured, the overhead of sharing IRQ vectors is to be avoided. It is also clear that additional real servers will not help if your director does not have the processing power to handle the extra load. One may want to consider adding a CPU to the director after the number of real servers has also been increased (this ratio was close to 1 to 1 in our testing).

One of the most valuable lessons to take away from this project is the ability to generate a high work load with a limited number of client machines. Our initial attempts were throttled by system resource limitations so that we could not open more than 1024 concurrent connections. After a substantial debugging effort, later tests we were able to increase this to approximately 5500 concurrent connections,[8] even on our smallest client (see Figure 2).

A key asset of LVS is the willingness of the open source community to enhance it and respond to the questions of users via the LVS mailing list. We have found it to be an invaluable resource that has assisted our efforts many times.

Acknowledgments

Mission Critical Linux would like to thank Intel Corporation for their assistance in getting access to various hardware and software resources, especially John Baudrexl, Vikram Saletore, Mike McCardle and Greg Regnier. They provided invaluable assistance to MCLX in scoping our experiments as well as supplying feedback throughout the project. Kris Corwin of MCLX assisted the authors in diagnosing several issues related to httperf and the e1000 device driver. Joseph Mack, Wensong Zhang and Julian Anastasov graciously reviewed an initial draft and offered excellent comments. David Mosberger assisted in our efforts to increase the ceiling on the number of connections a single client could achieve with httperf. Finally we would also like to acknowledge the members of the LVS mailing list for providing timely responses to our inquiries and for being an invaluable source of information concerning LVS.

Author Information

Pat O'Rourke worked as a kernel engineer at Mission Critical Linux, Inc. where he was responsible for adding LVS support in MCLX's Convolo Netguard cluster product. He is currently employed at Egenera, Inc. working on their BladeFrame server. He can be reached at porourke@world.std.com.

[8] As the numbers are reported by httperf.

References

[1] Zhang, Wensong, "Linux Virtual Servers for Scalable Network Services," http://www.linuxvirtualserver.org/ols/lvs.ps.gz.

[2] Zhang, Wensong, "LVS Deployment page," http://www.linuxvirtualserver.org/deployment.html.

[3] Mack, Joseph, "The Linux Virtual Server HOWTO," http://www.linuxvirtualserver.org/Joseph.Mack/LVS-HOWTO-991205.gz.

[4] "LVS Documentation," http://www.linuxvirtualserver.org/Documents.html.

[5] Foundry Networks, "ServerIronXL vs. Alteon 180e: Layer 4 Maximum TCP/IP Session Rate and Concurrent Session Capacity," http://www.foundrynetworks.com/testreports.html#ServerIron.

[6] Freed, Les, "How We Tested: Load Balancers," *PC Magazine*, http://www.zdnet.com/pcmag/stories/reviews/0,6755,2455845,00.html.

[7] ArrowPoint Communications, "Testing Stateful Networking Devices for Web Applications," http://www.arrowpoint.com/solutions/white_papers/product_testing.html#howto.

[8] Snell, Quinn O., Armin R. Mikler, and John L. Gustafson, "NetPIPE: A Network Protocol Independent Performance Evaluator."

[9] Anastasov, Julian, "testlvs," http://www.linuxvirtualserver.org/software/index.html.

[10] Mosberger, David, and Tai Jin,' "httperf – A Tool for Measuring Web Server Performance."

[11] eTesting Labs, ArrowPoint Communications, http://www.zdnet.com/etestinglabs/stories/main/0,8829,2462091,00.html.

[12] Mack, Joseph, "LVS Performance, Initial Tests with a single Real server, LVS," http://www.LinuxVirtualServer.org/Joseph.Mack/performance/single_realserver_performance.html.

Measuring Real World Data Availability

Larry Lancaster and Alan Rowe – Network Appliance, Inc.

ABSTRACT

This paper examines how server marketing claims of high reliability (e.g., 99.999%) stack up against real world data measurements. Our goals were to: measure discretionary NFS data availability, compare data availability between standalone and clustered systems, and draw some conclusions about best practices for customers.

We explain our methodology for measuring, filtering, and categorizing availability-related data. Through careful data and error analysis, we arrive at discretionary NFS data availability estimates for NetApp filers in the real world. We conclude that NetApp clusters provide over four-nines availability in the field.

Introduction

The server marketing world is full of claims of high availability. These claims usually take the form of "3-nines" (99.9% availability), "4-nines" (99.99% availability), or even "5-nines" (99.999% availability) [Rich99]. One is left to wonder at how these claims compare to real-world availability.

Vendors are known to partner in a marketing relationship with the purported goal of providing 5-nines availability. However, even then the "guaranteed" availability can be much lower [HP01], and the quantitative basis on which the validity of the guarantee should be predicted or disputed remains unclear.

Among compute server vendors recently, Microsoft commissioned the Aberdeen Group to perform a study which was based upon real-world data [Aber01]. A Microsoft-developed system monitoring tool collected outage data from systems at various Microsoft customer sites.

Data from 10 companies was considered in the Microsoft study, for a total of about 330 observed system-years. The result was that the hand-picked set of customers were seeing about 99.96% availability in the field. Whether and on what basis customers, systems, and/or outages were excluded remains unclear.

Among dedicated file server vendors, however, we could find no recent analogous study of real-world data availability. While similar studies may have been commissioned and carried out, they may not have been made public. We believe that a sound empirical basis for studying file server data availability is necessary to establish, track, and refine product quality.

Since October 1999, Network Appliance has been shipping versions of DataONTAP (our proprietary file server OS) with availability metrics support. This development has allowed us to perform a trailing-year analysis of NFS data availability across all systems running the appropriate releases and configured to send the data to Network Appliance.

Goals

Our goals for this study were threefold. First, we wanted to reliably measure discretionary NFS data availability,[1] within the complex and uncontrolled environment of the real world. We define *discretionary availability* for a product-service (the product being a NetApp filer and the service being NFS file service, in this instance) as one minus the fraction of time the service is unavailable due to the failure of any one or more component(s) of the product.

Discretionary availability is not impacted by intentional downtime or downtime due to external power failures, operator blunders, installation, testing, or moving. By "the real world" we mean the distribution of NetApp customer environments. This distribution includes countries and localities with poor power distribution, localities far away from parts depots and Customer Support centers, and customer testing facilities that aren't always knowable to us. We believe there is no more meaningful laboratory in which to measure a product's discretionary availability than the distributed laboratory of the product's customer base.

Second, we wanted to be able to compare data availability between standalone and clustered systems. Clustered configurations have been proposed and implemented for both general purpose servers [Barber97] and specialized file server appliances [Kleiman98]. The purpose of such an architecture is to provide high availability (HA) through failover capability. We wanted to see if the prediction of HA for clustered configurations was realized in real-world environments.

And third, we wanted to be able to draw some conclusions about best practices for customers. This is an ongoing study for us, but we wanted to gather at least some basic information immediately.

Methodology

Real-World Data Collection

DataONTAP has a feature called Autosupport. When enabled, the file server will send an email whenever something noteworthy happens (including after an outage), and weekly no matter what. The

[1] We support both NFS, CIFS, and multi-protocol NFS plus CIFS. For our first study, we chose systems running at least NFS. Other studies are ongoing.

customer can choose to have emails sent back to Network Appliance and/or internally within his/her own company.

If they're sent to Network Appliance, and they're from systems running recent enough releases to give us the availability-related ("availtime") data we need, they become part of our sample pool of raw emails.

SMTP is not an entirely reliable transport. Even if it were, there are sometimes network problems or filer problems surrounding an outage, such that an email may not be properly generated or received at or around that time. We needed to design a data collection and reporting system which would work around these kinds of difficulties.

Availtime Deltas

Availtime data consists of a set of cumulative counters. Relevant to this study are the cumulative seconds since NFS was first licensed, and the cumulative seconds of unplanned NFS downtime since NFS was first licensed.

Because availtime counters are cumulative, the availtime information from consecutively received pairs of emails can be differenced to tell how much time passed between generation of the emails, generation of the emails, and during how much of that time the service was unavailable. The resulting pairs of differences are called "availtime deltas," and form the basis for a rigorous event-based model of availability. Each delta is associated with a pair of consecutively received emails from the same system.

Logs and Configuration Information for Diagnosis

In addition to availtime data, autosupport data contains detailed configuration information and cumulative system logs to help us automatically diagnose system failures, if any, associated with an outage or a set of outages. Those emails can be examined to see what was reported in the portion of the log new from one received email to the next, and whether any configuration information changed over that period of time.

Because system logs are cumulative and are regularly sent to Network Appliance on a weekly basis, clues to the cause of an outage will still be available in the log when the next email is received. System configuration information between consecutively received emails can still be compared on a before-and-after basis to determine if parts were swapped or other configuration items were changed.

Data Filtering

In order to measure discretionary availability as defined in the previous section, we need to filter out intentional outages, installation, testing, and moving. We also need to be able to deal with garbage data.

To filter out intentional outages, we consider all *unplanned* NFS outages in this study – that is, outages where the operator did not intentionally reboot or halt the machine, unless he specified that a core be generated (a diagnostic reboot or halt is considered unplanned). This distinction is made by the filer in recording the availtime data, so that no post-processing is required.

An unplanned outage is a serious event. Simple failures do not cause these. For example, a single disk failure just causes a spare disk to be assigned to the RAID group instead; memory and NVRAM has ECC logic; systems have dual power supplies and multiple fans; single motherboard or adapter failures can be dealt with by clustering; disk Fibre Channel loop failures can be handled by dual-pathing the disks, and so on. That's why it's important for us to examine data relating to all unplanned outages, in order to determine the cause of failure.

Next, we filter out garbage data with a set of basic sanity checks to insure that the availtime deltas don't defy reason. For example, setting the clock forward or backward by a year – something that happens more frequently in the field than one might at first imagine – might cause the total time difference to be either 53 or -51 weeks.

Filtering out installation, testing, and moving is a more difficult proposition. One step we take to address installation and related testing is to exclude from analysis availtime deltas from any system for the first 28 days after NFS is first licensed. This is hardly a sufficient measure. NFS might be licensed on a demo-eval unit, or as part of the manufacturing test process. To do a better job, we introduce the concept of an "availtime system."

Availtime Systems

An availtime system can be thought of as an extended system key which uniquely identifies a system with reference to where it is and how it is deployed. The components of the key include system ID, OS version, domain name, hostname, and clustered indicator (indicating whether the system is one node in a cluster). One unique combination of these values comprises an availtime system, and only availtime deltas between consecutively received emails from the same availtime system are included for analysis.

The domain name and the hostname insure that systems which are installed, are moved, or change owners don't generate input to the analysis. Changes in the other keys help prevent the inclusion of availtime deltas related to testing of new system configurations.

Testing is not completely screened out by this filtering process. Some customers are known to have dedicated machines just for the purpose of testing, and we have no systematic way of excluding these systems.

Other Sources of Availtime Data Exclusion

We do not filter based upon customer, system, location, filer model, release, or system management practices for the purposes of this study. The real world

is a rough and nasty place. Literally, if a filer died and the customer decided "it's Friday, I'm tired, I'll fix the problem on Monday," a three day outage would be included in this study.

All machines are not equivalent products, or in identical settings, other than being NetApp filers serving NFS. All eligible availtime deltas from all eligible availtime systems are included in this analysis. As such, the results of this study apply as generally as possible to our products and our customers.

It is possible that a customer chooses not to enable the Autosupport mechanism, or does not run a DataONTAP release recent enough to support availability analysis. In either case, we obviously cannot include their data in this study.

Failure Categorization

In order to measure discretionary availability as defined in the previous section, we need to be able to determine which outages fall into the categories of operator error and power failure, and which were specifically related to the product-service. We took as a starting template for outage categorization a study done by Jim Gray [Gray90].

Our study differs in some important ways, though, so we modified our outage categories as appropriate. The categories we use are disk subsystem hardware, non-disk subsystem hardware, software, operator, power, and likely power failure.

Our algorithms for attributing the cause of an outage to one of these categories include scanning the cumulative system logs, examining the system configuration information before and after the outage, and looking for site-wide events by correlating availtime deltas and system-logged information across multiple systems at the same site.

When scanning the system logs we can find each of the downtime events within the log, and look both back (earlier) and forward (later) in the logs from that point to find more information about the outage. When examining configuration information from before and after the outage, we can determine if the customer had misconfigured something, or if a part was swapped out during that time. When correlating events across multiple systems at the same site, we can determine that a site-wide outage took place.

Examples of Outage Classification

Disk subsystem hardware failure is a category of failure which affects access to multiple disks in the same RAID group. Typical causes include loss of an adapter or cable or one of the disk shelf circuits in a non-clustered or non-dual-pathed configuration. We can detect this by looking for messages in the system log which indicate failure of the loop, such as:

```
Wed Oct 11 09:32:47 GMT [isp2100_timeout]:
Offlining loop attached to Fibre Channel
adapter 4.
```

Non-disk subsystem hardware failure includes all other hardware failures. We can detect these failures by syslog messages which are specific to hardware failures; for example, a bad (and uncorrectably bad) memory chip:

```
Fri Oct 13 13:45:14 GMT [callout]:
1 UNCORR PROC ERR 86 eia:87a0000000
fs:005d eis:c5000000 isr:100000000 ...
CPU ECC error on DIMM J28 at
address=0xa000000f, bit=39.
```

Software failure is visible when we examine the logs and detect either that the software panicked, or that the administrator deliberately created a core file:

```
Fri Feb  9 01:44:16 GMT
[savecore_admin:info]:saving 1048M
to /etc/crash/core.0.nz (assertion
"!sk_is_mp_mode || (sk_n_domains == 2
|| _sk_own_domain(procp->domain))"
failed . . .)
```

Power failure consists of two different categories. One is **definite** power failure, which we can detect by observing that multiple systems suffered a dual power supply failure at the same site at about the same time. The other is a **likely** power failure, which we can recognize by noting that both power supplies failed on the system at about the same time.

Operator failure includes outages due to mis-administration. Our system is an appliance and so is much less susceptible to mis-administration than the general-purpose computers noted in [Gray90]. However, operator failures do exist in real-world data.

For example, one way we can detect an operator failure is to note that both power supplies failed on a system at about the same time, and once the system came back up, parts unrelated to the power supplies had been swapped. This observation indicates that an administrator pulled the plug on a machine in order to swap hardware, without first performing a clean shutdown.

Waiting for Parts and Customer Deferral

In order to measure discretionary availability as defined in the previous section, we need to be able to determine how much of the hardware- and software-related downtime as measured using the availtime data gathered through Autosupport is specifically related to the product-service.

There are two significant downtime factors which cannot be measured through Autosupport-derived data, but which must be factored out to estimate discretionary data availability. The first is the amount of time spent waiting for parts to arrive at a customer site. The second is the amount of time the customer chooses to wait because he/she regards the outage as uncritical, and taking care of the problem immediately would be inconvenient.

All outage-related data from our Customer Satisfaction department relating to the same physical

systems and the same trailing year period as the Autosupport data were thoroughly examined and audited over a six-month period, case by case, to determine the portions of downtime that were attributable to product failure and diagnosis, customer deferral, and waiting for parts.

This data was collected and audited in a painstaking manual process, and so can not be mapped one-to-one with availtime events. However, the overall ratio is the best available estimator for the product-related proportion of downtime. We can use this result in conjunction with the availtime-based product-related availability figures to estimate discretionary availability.

Data and Analysis

Categorization of Empirical Availtime Data

Over the trailing year ending April 28, 2001, the availtime deltas forming the sample pool for this study spanned 4400 observed system-years. We summarize the empirical data by showing the relative impact on availability from each of our downtime categories: disk subsystem hardware, non-disk subsystem hardware, software, operator, power, and likely power failures. Since one of the goals of this study was to look at the differences in discretionary availability between systems in clustered and standalone configurations, we broke out the data along these lines, as well.

We compare results for the entire sample base against results for a "select" subset of systems, thought to be well-managed production systems at large, international, well-known customer sites each with 100 or more systems in both standalone and clustered configurations. We also compare these results against results for one particularly large customer, whose site we consider to be the best-managed overall, with frequent onsite visits from NetApp Customer Support representatives. The empirical data are summarized in Figure 1.

Error Analysis

To continue forward and obtain meaningful results, we need a basic model of the data to determine how to proceed. In particular, it is our goal to estimate discretionary filer-NFS data availability for the population of Network Appliance filers, in both clustered and standalone configurations, and to have some idea of the uncertainty in these estimates.

Our sample pool of availtime deltas contains about 4400 system years worth of data. However, these data are reported by only a small portion of our entire customer installed base, on a self-selecting basis as described in the previous section. And, we're only looking at these systems for a one year period. We'd like to be able to estimate the mean availability across the population of either standalone or clustered filers, at large and in general. As discussed in [Burgess00], we need a measure of standard error (SE) to tell us how accurate our estimates are.

There are a number of sources of measurement error in our estimates, as well. By virtue of the way

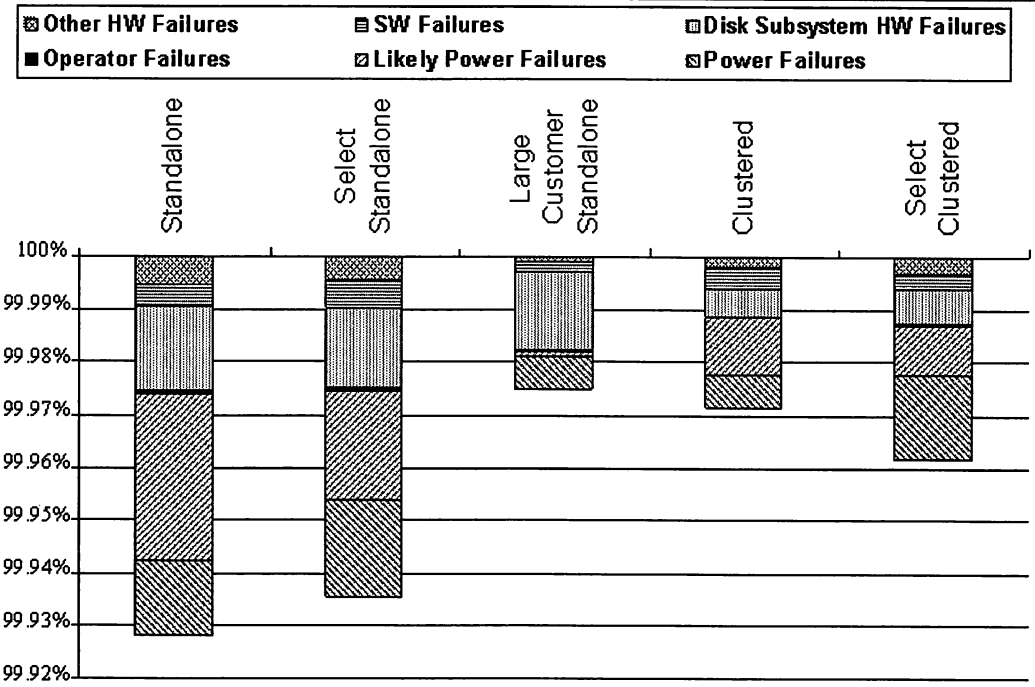


Figure 1: Empirical data categorization: availability impact by category. This graph shows the results of our outage categorization, and the overall impact of each category on the measured availability. We were surprised to find that operator failure accounts for a trivial portion of overall downtime. Note the size of the power failure categories: a UPS is worth the expense, if the goal is an HA environment. Note also the dramatic decrease in HW-related downtime when moving from a standalone to a clustered configuration.

DataONTAP availability metrics support is designed, for example, the tabulated availtime deltas have an unbiased ten second granularity. Also, as described in the previous section, our estimate of the product-related portion of downtime – relative to waiting for parts and customer deferral – was obtained by hand on a case-by-case basis, making it subject to error and rounding. All of these sources of random error can be addressed with a SE, as well.

However, calculating a meaningful SE on an estimate of population availability is much more difficult than one might at first imagine. We know that the availtime deltas which comprise our sample pool are not independent and identically distributed (IID). The availtime deltas contributed by any one system tend to be highly dependent among themselves.

Empirical outage durations and availabilities are far from normally distributed, both at the availtime delta level and when rolled up to the availtime system level. We have very little in the way of a parametric model for calculating uncertainty in our estimate. Taking these facts together, there is no quick and easy formula to obtain SEs for our population estimates that would be valid in this case.

The Bootstrap

We turn to a bootstrap [Efron93] resampling procedure to calculate our SEs. With a very basic and sparse set of assumptions, we can reasonably compute the SEs of our measurements.

Recall that an availtime delta is a vector containing both the difference in total time and the difference in downtime between consecutively received Autosupport emails. Let D_i represent the sum of all availtime deltas for one availtime system over the course of the observation period. Here, $(i: 1 \cdots N)$ indexes the N availtime systems comprising our sample pool, S_0. The sample availability parameter of interest $A(S_0)$ is calculated as:

$$A_0 = 1 - \frac{\sum_i Ddown_i}{\sum_i Dtotal_i}$$

where $Ddown_i$ is the downtime component of D_i and $Dtotal_i$ is the total time component of D_i.

Assume that the D_i are random vectors IID w. r. t. some unknown probability distribution P. (This assumption is strongly supported by the qualitative similarity of the empirical hardware + software data availability components between the "select" systems and "all" systems, as seen in Figures 1 and 3.)

The bootstrap resampling procedure, then, is as follows. Construct a sequence of new sample pools, S_j, indexed by $(j: 1 \cdots M)$. Each S_j is created by sampling N of the D_i at random, with replacement, from S_0. For each S_j, compute the sample availability parameter, $A_j(S_j)$. Then, compute the unbiased SE of the $A_j(S_j)$ over j. The bootstrap lets us substitute the resulting SE (as N and M grow large) for the SE which would be seen if the S_j were drawn from P rather than from S_0.

Using the same basic technique with minor modifications, we can derive SEs for the portion of

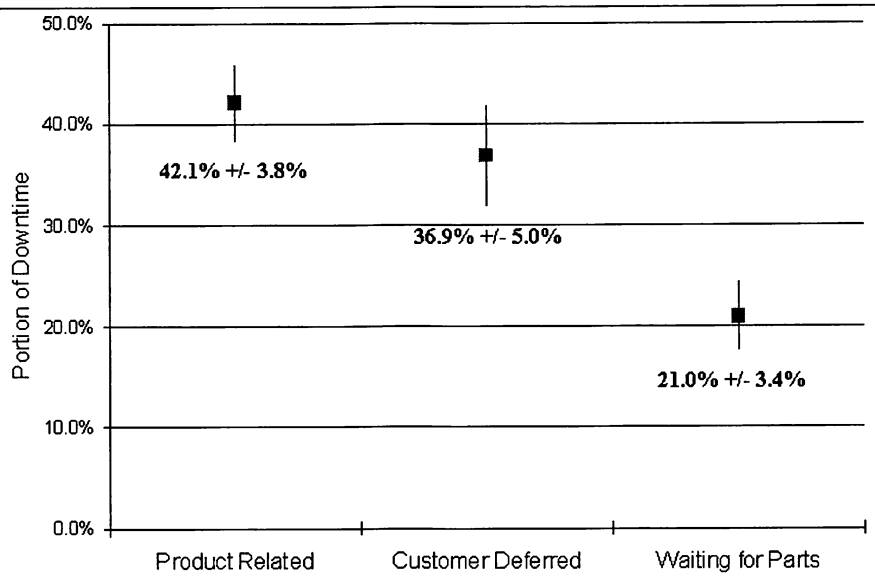


Figure 2: Portions of downtime due to product failure, waiting for parts, and customer deferral. This graph shows a categorized distribution of downtime, based upon a study of Customer Support case records over the same period of time as the rest of this study. Bootstrapped SE bars are also included (M = 5000 resamples). These results are used to obtain the conditional probability that a unit of downtime is product related, for the purpose of estimating discretionary availability. Note the large amount of "Customer Deferred" downtime: any field study of availability will be confronted with factoring this out. Also note the size of the "Waiting for Parts" category: this shows that customers would do well to keep a stock of commonly-used parts on-hand.

availtime-calculated downtime which is actually due to waiting for parts and customer deferral (see Figure 2).

The bootstrap is not without its weaknesses. For example, outliers captured in S_0 will tend to exaggerate the calculated SE. But the effect of this problem is to understate, rather than overstate, certainty in the estimate of $A(P)$. In other words, it leads to more conservative reporting. We wholeheartedly recommend the bootstrap technique to those seeking a measure of uncertainty around estimates when a classical determination of such a measure is not readily available.

Discretionary Data Availability

Beginning with the empirical data summarized in Figure 1, we factor out power and operator failures, leaving all hardware-and software-related outages. This extraction takes us as close as possible to an estimation of discretionary availability using availtime data alone.

We then apply the bootstrap resampling procedure to compute SEs (M = 5000 resamples), and display the results in Figure 3. We already have the results of the Customer Support analysis of waiting for parts and customer deferral in Figure 2, also with bootstrapped SEs (M = 5000 resamples). It remains only to combine these results into a cohesive whole.

We proceed as follows: the availtime-derived availability A_a, as shown in Figure 3, is equal to $1 - P(down)$, where $P(down)$ is the probability that a system-service will be down in any small sliver of time due to product failure, waiting for parts, or customer deferral. The probability that a system-service is down in any small sliver of time due to product failure, given that the system is down, is shown by the first data point in Figure 2: call this $P(prod|down)$.

What we want is the discretionary availability A_d, which is given by $1 - P(prod)$, where $P(prod)$ is the probability that a system-service will be down in any small sliver of time due to product failure. Taking these probabilities as independent, Bayes' rule tells us that $A_d = 1 - P(prod) = 1 - P(down)P(prod|down)$.

To get an SE for A_d, we notice that $SE(A_d) = SE(P(down)P(prod|down))$. We can then solve for $SE(A_d)$ by using the well-known technique for propagating SEs of two independent random variables into their product, so that

$$\frac{SE^2(A_d)}{E^2(1 - A_d)} \approx \frac{SE^2(A_a)}{E^2(1 - A_a)} + \frac{SE^2(P(prod|down))}{P^2(prod|down)}.$$

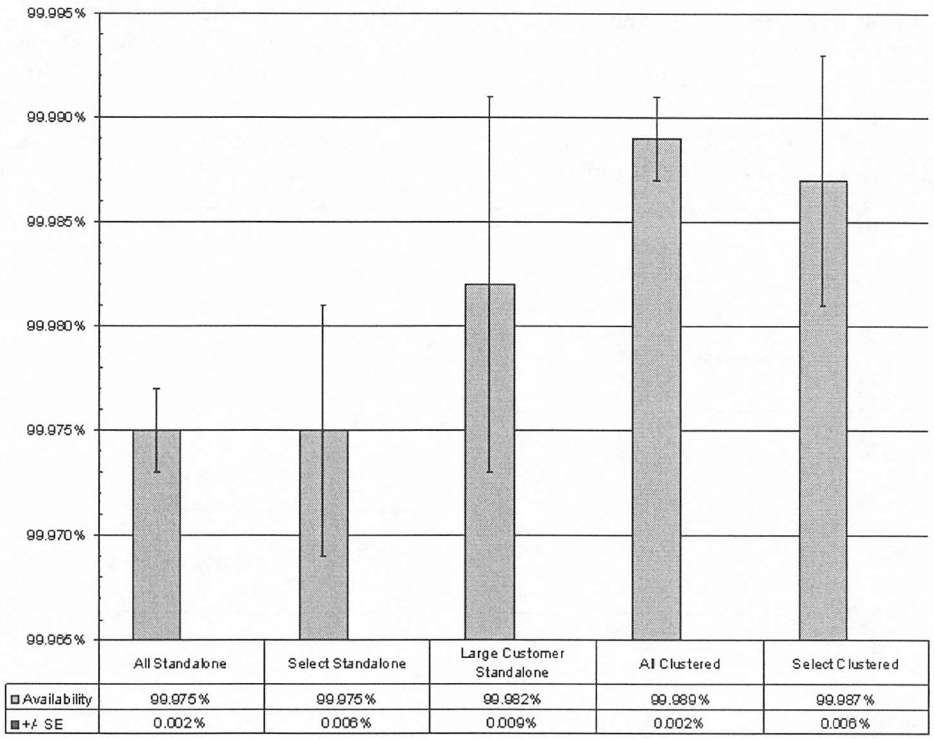


Figure 3: NFS data availability from availtime data (includes waiting for parts and customer deferred downtime). This graph shows the estimates of NFS data availability gotten from availtime data alone, for different sets of availtime systems. Bootstrapped SE bars are also included (M = 5000 resamples). This graph excludes the operator failures and power outages depicted in Figure 1. Note the overlap of the "All" and "Select" categories, for both clustered and standalone configurations. This implies that data availability can reasonably be considered a property of the population. Note also that the difference between aggregate clustered and standalone configuration availabilities is highly significant. This difference demonstrates that clustering is an effective tool for achieving HA.

Here, $E(.)$ denotes the expectation (average). The results are displayed in Figure 4. This completes our estimation of real-world filer-NFS discretionary data availability.

Conclusions and Direction

It is possible to track discretionary product-service availability in the real world: We believe that the similarity between the results for all sites and the results for "select" sites in Figure 4 also demonstrates that the procedures we're using to generate, collect, filter, and aggregate the data are giving a fair and objective view of discretionary availability.

NetApp clusters provide 99.99% availability: Figure 4 shows that nodes in a clustered configuration provided greater than 4-nines discretionary NFS data availability in the field. Nodes in a standalone configuration are shown to be within the margin of error of 4-nines, as well.

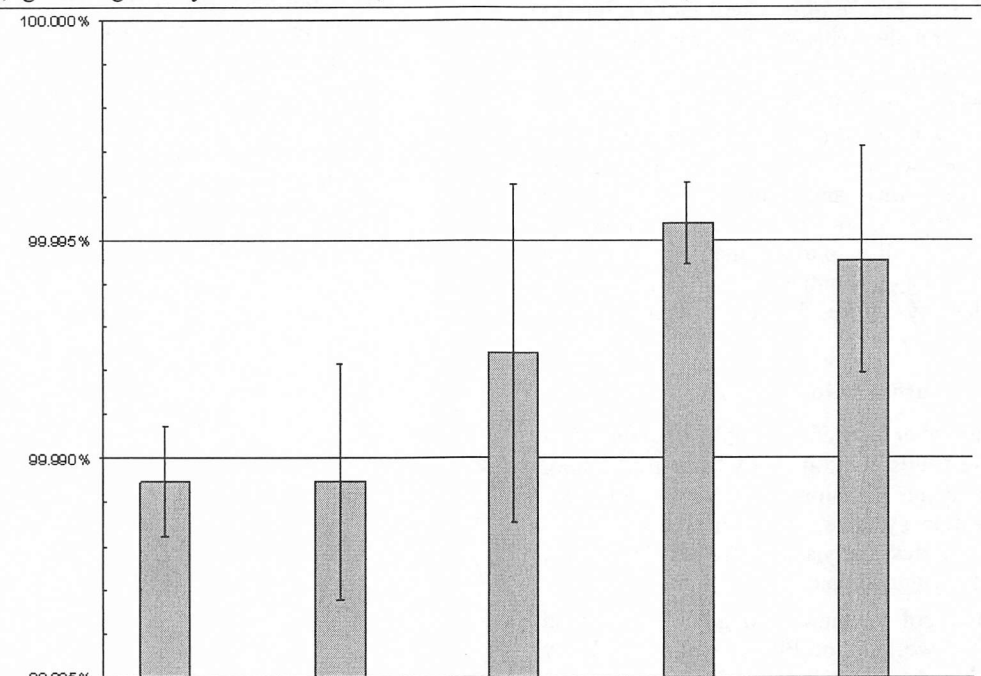
A clustered configuration does provide higher data availability in the real world: As Figure 4 shows, clustered systems provided significantly better availability than standalone systems. A common rule-of-thumb for determining whether two sample means are different, given rigorously measured SEs, is to check whether the error bars of the two measurements are mutually exclusive. In this case, they are separated by this distance twice over.

This observation fits with prediction, since the point of a cluster is to allow one head to serve data for its own and another data node while the second head is down. This results in only a brief non-discretionary outage during takeover, during which NFS file service would be unavailable.

Redundancy of paths to disks also provides higher data availability. All clustered nodes have that while most unclustered nodes do not, since the feature was only introduced within the timeframe measured here.

All customers can achieve enterprise-level availability: As Figure 4 shows, and as we were surprised to find, availability is not significantly separated between "select" systems and all systems in either a clustered or a standalone configuration. We believe that this uniformity reflects the ease-of-use and productized reliability that are goals of the appliance concept.

Figure 1 supports this notion. Operator failure represents a trivial portion of the overall downtime, suggesting that ease-of-administration is key to high data availability in the real-world.



	All Standalone	Select Standalone	Large Customer Standalone	All Clustered	Select Clustered
Availability	99.989%	99.989%	99.992%	99.995%	99.995%
+/- SE	0.001%	0.003%	0.004%	0.001%	0.003%

Figure 4: Filer-NFS discretionary data availability: population estimates. This graph shows the estimated filer-NFS discretionary data availability, for different sets of availtime systems. The values are derived from the data shown in Figures 2 and 3. SE bars are also included, derived from those in Figures 2 and 3 using classical error propagation analysis. Note that the difference between aggregate clustered and standalone configuration availabilities remains highly significant after adjustment for discretionary outages. This difference demonstrates that clustering is an effective tool for achieving HA.

Power failures are common – use a UPS: As Figure 1 shows, power failures are commonplace in the field. We believe that a UPS is a critical component of an HA environment.

Parts failures can cause unnecessary wait – keep a stock of parts: Availability afforded the large customer depicted in Figures 1, 3, and 4 is qualitatively higher than that afforded other customers running standalone configurations, although the margin of error is high due to the relative size of the sample.

This large customer subscribes to the NetApp Global Advisor level of support. The advisors for this customer have recommended that the customer keep a large stock of spare parts on-site. Because about one-third of non-discretionary downtime can be attributed to waiting for parts (see Figure 2), we believe that one significant factor in this customer's success has been to avoid this delay.

More Global Advisor recommendations have evolved and spread to other sites since April 28, 2001. We look forward to examining the effects of these measures on availability in the year to come.

Future Direction

We have only just begun to make sense of this vast wealth of data. Future efforts will include further characterization of the causes of downtime at the availtime event level and enhancements to the information sent back to NetApp through Autosupport.

We also look forward to examining the effects of various measures we take to help improve discretionary data availability, and widening our scope to include various types of planned outages. Ultimately, we hope this study will help to change the competitive landscape by forcing our competitors to compete using rigorous availability figures based on data, not just hype

Author Information

Larry Lancaster earned a couple of degrees at Berkeley in the 1990s, took an LOA from grad school in 1998, worked for Decision Focus developing revenue maximization systems, and is now a tools programmer and metrics analyst at Network Appliance. Reach him at larryl@netapp.com .

Alan Rowe got a couple of degrees in Scotland in the late 60's, worked for Plessey on a capability machine, emigrated to Canada to work for Bell-Northern on the DMS digital switch, emigrated to the US to work for Tandem on the NonStop fault-tolerant system, and is now Technical Director at Network Appliance, where he cares about Available, Simple, and Fast file server appliances. He plans no further emigrations. Reach him at rowe@netapp.com.

References

[Aber01] Aberdeen Group, Inc., *Proving-the-Point: Interviews with Next – Generation Windows 2000 dot.com*, Microsoft Executive White Paper Updated – for Electronic Redistribution Only: http://www.microsoft.com/windows2000/server/evaluation/news/reviews/dotcoms.asp, Redmond, Microsoft, Feb. 6, 2001.

[Barber97] Barber, M. R. "Increased Server Availability and Flexibility through Failover Capability," *Proceedings of the Eleventh Systems Administration Conference*, Berkeley, USENIX Association, p. 89, 2001.

[Burgess00] Burgess, M., *Principles of Network and System Administration,* Chichester, Wiley, 2000.

[Efron93] Efron, B., R. J. Tibshirani, *An Introduction to the Bootstrap*, New York, Chapman and Hall, 1993.

[Gray90] Gray, J., *A Census of Tandem System Availability: 1985-1990,* Tandem Computers TR 90.1, January, 1990.

[HP01] Hewlett-Packard, Inc.,"Helping Companies Across the Globe Deploy Highly Available Solutions," HP-UX High Availability Software Home (as of June 4, 2001), http://www.hp.com/products1/unix/highavailability/5nines/5n5mbrief.pdf.

[Kleiman98] Kleiman, S. R., S. Schoenthal, A. Rowe, S. H. Rodrigues, A. Benjamin, "Using NUMA Interconnects to Implement Highly Available File Server Appliances." *Proceedings Hot Interconnects*, Vol. 6, August, 1998, and *IEEE Micro*, Jan./Feb. 1999.

[Rich99] Richards, D. "EMC drops a container load of products onto the storage market," *Computer Reseller News*, http://www.itnews.com.au/crn/news/008_0304i.htm, Mar. 11, 1999.

Simulation of User-Driven Computer Behavior

Hårek Haugerud and Sigmund Straumsnes – Oslo University College

ABSTRACT

We simulate a computer system by modeling its users as individuals who have separate needs for resources like processes and login time. During the simulations the users make decisions with probabilities which depends on the time of day and on the character of the user. This makes us able to reproduce the large scale behavior measured at real computer systems as well as predicting the behavior of systems when varying the number and characters of the users.

Introduction

One of the goals of system administration is to be able to understand the characteristic interaction between humans and machines more fully, in order to use such knowledge to improve the reliability and security of computer systems [1]. To understand this interaction, one must blend empirical data with models and simulation, thus relating cause to effect [2]. A full understanding of computer systems requires a knowledge of important scales and processes. This is far more than can be achieved with a single model or study; however, at the level of the user-computer interaction, high level changes in the system generally occur over intervals greater than several minutes. This leads to a considerable simplification.

Recently some progress has been made in trying to characterize the state of large and complex computer systems, in terms of a few macroscopic parameters [3, 4]. This work is based on measuring at intervals of a few minutes, a large number of system variables like the number of users and processes, network connections, memory and disk usage etc. One of the goals has been to be able to measure the normality of a computer system, making it possible to automate the detection of abnormal behavior. Burgess has provided a stochastic explanation for the subset of variables which closely follows the periodic trends of users [5]. This approach uses functional methods of statistical mechanics. However, the approach is limited to fairly simple calculations, and there is room for a more general way to predict system behavior, by Monte Carlo simulation.

The model presented here attempts to simulate the behavior, in time, of a selection of variables which describe resources on the system, by imitating the behavior of the users themselves. Previous work has been done to model some of this behavior by assuming that large numbers of users behave more or less like a mass crowd (a gas of users) [5]. Here we start with some simple assumptions about the probable behavior of individual users and repeat this behavior for a given number of users. This allows us to adjust the number of users and see how behavior depends on various parameters. This is potentially more complicated than the analytical analysis in [5], but it is more general and flexible for extending that work.

A simulation environment is ideal for testing such theories, since it becomes easy to change conditions which are impossible to change on real systems. A simulation is also an efficient tool for predicting the expected load and resource usage of a system when the number of users or the system itself changes. The functional analysis of Burgess is only true in the limit of large numbers of users, and thus must be understood mainly as an idealized limit on behavior.

Empirical Data

Earlier empirical work came to two main conclusions about the behavior of macroscopic system variables: that many (though not all) variables are periodic in nature and that the behavior was stochastically distributed around stable averages, in the manner of a steady state, modulated by periodic variation. These empirical facts can be seen in the periodogram of Figure 1, which plots a number of these variables, scaled over the course of a week.

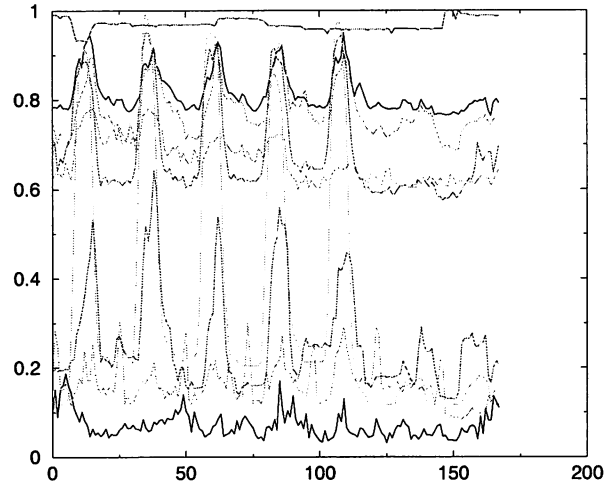


Figure 1: A trace of several scaled system variables like number of users and processes, free diskspace, www connections, etc., over the course of a week.

Traditionally, measurements of normal computer behavior have been used to analyze event arrival times and lifetimes; they have often been collected in connection with performance analyses [6, 7, 8, 9, 10]. Other studies of computer systems have been performed in connection with load balancing [11, 12, 13, 14], expectations of communications over a network [15, 16] and interactions with users on teletype terminals [17]. Our primary aim here is to show how a simple random model can be used to understand system behavior, and to test predictions about management decisions. In the classification used by Burgess [1], this a type I model.

Simulation

In a Monte Carlo simulation we attempt to follow the time dependence of a model for which change does not proceed in some rigorously predefined fashion but rather in a stochastic manner which depends on a sequence of random numbers which is generated during the simulation. With a second, different sequence of random numbers the simulation will not give identical results but will yield values which agree with those obtained from the first sequence to within some statistical error.

The simulation presented here models users of a computer system as individuals who make decisions in a stochastic manner, but far from completely at random. It is for instance more likely that a normal user logs on to the system in the morning than in the middle of the night.

The simulation is object-oriented and written in C++. The main objects are the users, characterized by their individual needs for resources, and the hosts with a given set of resources. The system one wants to simulate is devised by providing the number of users and hosts and their properties. The simulation is divided into time-slices and at each step, every user makes decisions such as starting a new process, logging out or using the disk. These decisions are made using random numbers to simulate Bernoulli trials, with probabilities which are time-dependent and reflects social behavior and work rhythms of the users [3] [18] [19]. A Bernoulli trial [20] is an experiment that either fails or succeeds and the probability for success is given by p. Flipping a coin is a Bernoulli trial with probability $p = 0.5$. Suppose that a user of the simulation has got a probability of $p = 1/6$ for logging out at some time of day. This corresponds to a user throwing a die every minute and once he gets a "six" he will log out. On average such a user will stay logged in for six minutes. A user who just pops in to check his email is in our simulations given logout-probabilities of similar magnitude.

The probability for logging on at, for instance, Sunday night is small and is much larger on Monday morning. In addition, the probabilities depend on the particular character of the user. Several characters can be included in the simulation, such as users who start their day early, users who mostly work at night, or those who just use their account for reading email once in a while. A percentage of these various user-characters is input to the simulation. Each user in each character-group differs slightly and they change somewhat from week to week within given boundaries.

The fundamental assumption of the simulation is that, at any given time, there exist probabilities for a user to log in or out, start a process and generally perform any action that influences the system. That such a probability exists is clear, but it certainly has a very complex dependence on time, the user's state of mind, and the state of the system. Making these probabilities only depend on time is a strong simplification, but it seems to grasp much of the correct dynamics of a computer system. Some limits given by the system, like a maximal number of users and processes or finite memory are straightforward to impose. One could also think of mechanisms which changed the user-probabilities with respect to how the system reacted (slow response, paging, slow network etc.) but this is beyond the scope of this short study.

The behavior of a computer system will obviously be strongly dependent on the kinds of users it serves. In the simulations we have several categories of users and initialize the simulation by simply assigning percentages of users from each of the following categories.

- system users (root, www, dns etc.)
- standard users working "9 to 5"
- users starting early
- users working late
- users checking email from time to time
- users who are logged in all the time or/and run batch jobs
- users who mostly work at night

A standard user has, for instance, a high probability for logging in between 8:00 and 10:00 on a weekday and a high probability for logging out after 5. The probabilities are never one or zero, so any action is in principle possible at any time.

A simulation is initialized by creating a new simulation-object which in turn creates all the users. The number and categories of users is globally defined and for each user a new user-object is created. The initialization method of the user-class then sets the time dependent probabilities for logging on and off and creating processes according to the character of the user. The properties of users within a character-group are also made slightly different so that no users are identical. Figure 2 shows the main objects of a simulation of a system containing 14 individual users on a single host which has a maximum number of 256 processes.

Time is split into time-slices of one minute as we are looking for the behavior of the system over relatively long time-scales. After all the users have been created the simulation runs by visiting all users in a

Round Robin manner and each of the users makes their decision according to their time-dependent probabilities for logging out or on, starting processes etc.

If the probability p for an event like logging out was kept constant with time, the number of failures before an event happened after a series of independent Bernoulli trials would follow the Geometric distribution [20]. The mean value of this distribution is $1/p$ and gives the expected time to elapse before the event occurs. The standard deviation for this distribution equals the mean value for small p's and both the mean value and the width of the distribution grows rapidly with decreasing p. The probability p for logging out is changing with time, but on a time scale of hours. So the probability distribution for a user event is split into several Geometric distributions with different p's, each lasting one or more hours.

The following is an example of how the state of a user is updated in a time-step. If a user is logged on and his probability for logging out is $p = 0.02$ at this time of day, a random number between 0 and 1 is generated and if this number is smaller than 0.02 the user is logged out. This simulates a Bernoulli trial with probability $p = 0.02$. the expected time for the user to stay logged on would then be 1/0.02, i.e., 50 minutes. At five o'clock in the afternoon the probability for a standard user to log out increases to $p = 0.06$ making it improbable that he will stay much longer than half an hour.

The various user-probabilities p were initially chosen based on the average amount of time $1/p$ which would run before the corresponding event would occur. Since these probabilities also depend on the time of day and the day of week, the expectation values are hard to estimate accurately. Some initial simulations where therefore run in order to tune the parameters to make "standard" users behave as expected, system users stay logged on more or less permanently, email checkers just pop in from time to time and not stay for days, etc. Finally the percentage of user characters where adapted in order to simulate a given system. A future improvement would be to be able to estimate model parameters directly from data obtained by measuring the user behavior at a system.

The user-probabilities of the simulations are summarized in Figure 3 and Figure 4. The actual probability at a given hour and day of week, is the product of the number in the row labeled by this hour and the number in the row labeled by this day. At the bottom of the tables, the probabilities for stopping and starting processes are given. Systems of various natures can then be simulated by changing the percentage of these users as shown in the next section.

Results

The simulations of user and process statistics has given results which are quite similar to previously measured real world results [3]. Figure 5 shows the

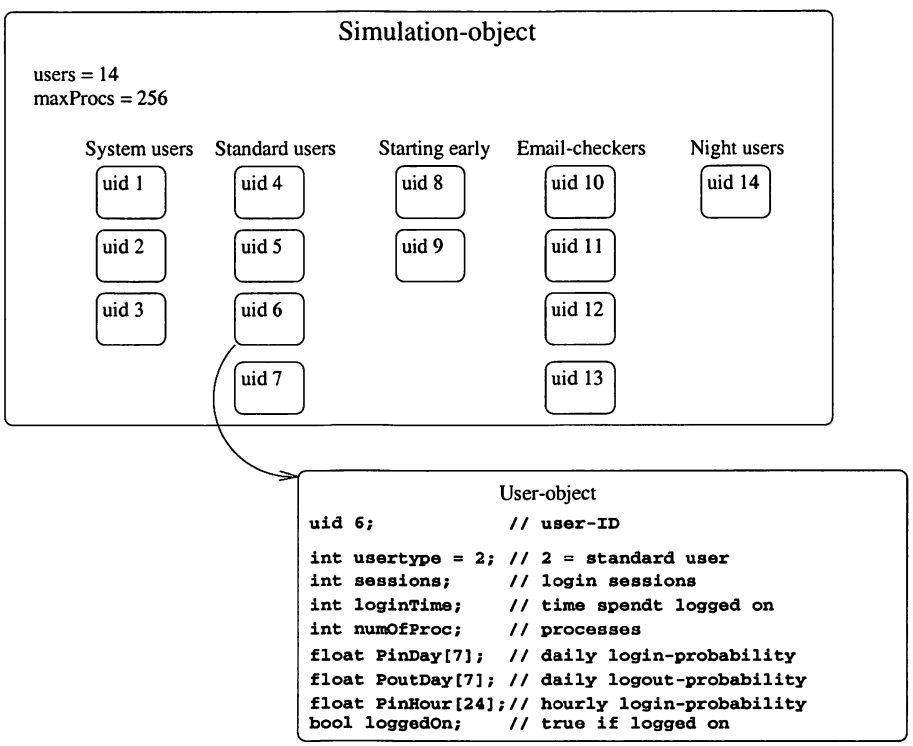


Figure 2: A simulation-object which contains 14 user-objects. Some of the variables of the user class are shown. A simulation step is performed by visiting all the user-object and calling their time-step method which performs actions according to the user's probabilities.

number of users logged on for a system of 100 users. The weekly rhythm is apparent.

The behavior of such a function obviously depends on the kind of users the computer serves. By changing the percentages of the different characters one can simulate computers with other user characteristics. In Figure 6 the simulation is carried out with a different composition of users; the main difference being the removal of the late night users.

In this way it is possible to tune the simulation to fit the data of a given computer system and afterwards predict what would happen when for instance the total number of users increased. Averaging over several simulation-years is done in minutes and makes it convenient to change and test the consequences of the kind of users who use the system.

Figure 7 compares the number of processes obtained from simulation and from measurement of a computer. The data are from the same computer and the same simulation as the data presented in Figure 5 are from.

In previous work [3, 4] the skew distributions of deviations around the mean value have been pointed out. We have obtained similar results in our

time/user	always in		system user		standard	
	in	out	in	out	in	out
Mon-Fri	1	0.001	1	1	1	1
Sat	0.1	0.001	1	1	0.1	1
Sun	0.01	0.001	1	1	0.01	1
0-1	0.0000	0.0200	0.0100	0.0003	0.0000	0.0200
2-5	0.0000	0.0200	0.0100	0.0003	0.0000	0.0200
6-7	0.0000	0.0200	0.0100	0.0003	0.0000	0.0200
8-9	0.0008	0.0040	0.0100	0.0003	0.0008	0.0040
10-11	0.0016	0.0040	0.0100	0.0003	0.0016	0.0040
12	0.0016	0.0060	0.0100	0.0003	0.0016	0.0060
13-14	0.0008	0.0080	0.0100	0.0003	0.0008	0.0080
15-16	0.0008	0.0080	0.0100	0.0003	0.0008	0.0080
17-18	0.0002	0.0400	0.0100	0.0003	0.0002	0.0400
19	0.0002	0.0800	0.0100	0.0003	0.0002	0.0800
20-21	0.0002	0.1600	0.0100	0.0003	0.0002	0.1600
22-23	0.0002	0.1600	0.0100	0.0003	0.0002	0.1600
stop	0.0263		0.0263		0.0154	
start	0.0263		0.0263		0.0263	

Figure 3: The probabilities characterizing "always in" users, system users and standard users working "9 to 5."

time/user	early		late		email		night	
	in	out	in	out	in	out	in	out
Mon-Fri	1	1	1	1	2	200	1	1
Sat	0.1	1	0.1	1	0.2	200	0.1	1
Sun	0.01	1	0.01	1	0.02	200	0.01	1
0-1	0.0000	0.0200	0.0002	0.0200	0.0000	0.0200	0.0003	0.0100
2-5	0.0000	0.0200	0.0000	0.0200	0.0000	0.0200	0.0003	0.0100
6-7	0.0016	0.0100	0.0000	0.0200	0.0000	0.0200	0.0003	0.0100
8-9	0.0008	0.0040	0.0000	0.0040	0.0008	0.0040	0.0001	0.0400
10-11	0.0016	0.0040	0.0000	0.0040	0.0016	0.0040	0.0002	0.0400
12	0.0016	0.0060	0.0000	0.0060	0.0016	0.0060	0.0002	0.0600
13-14	0.0008	0.0080	0.0008	0.0080	0.0008	0.0080	0.0001	0.0800
15-16	0.0002	0.1000	0.0008	0.0080	0.0008	0.0080	0.0001	0.0800
17-18	0.0002	0.1000	0.0016	0.0040	0.0002	0.0400	0.0024	0.0020
19	0.0002	0.0800	0.0016	0.0040	0.0002	0.0800	0.0024	0.0040
20-21	0.0002	0.1600	0.0016	0.0040	0.0002	0.1600	0.0024	0.0080
22-23	0.0002	0.1600	0.0008	0.0100	0.0002	0.1600	0.0024	0.0080
stop	0.0154		0.0154		0.0154		0.0154	
start	0.0263		0.0263		0.0263		0.0263	

Figure 4: The probabilities characterizing the users starting early, working late, checking email from time to time and users who mostly work at night.

simulations as seen in Figure 8. For each data point of the simulation, the deviation from the mean is calculated. The figure shows the number of data points as function of deviation from the mean, measured in units of standard deviation. The frequency of data points is counted by splitting the x-axis into 200 slots of equal size.

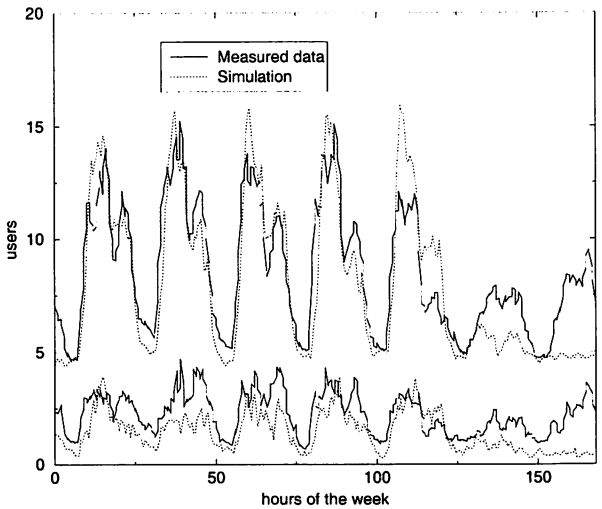


Figure 5: The number of users logged on as function of time; starting at Monday morning. The fully drawn line is the average number of users during a week at a computer at Kansas State University averaged over nine weeks. The dotted line are from a nine week long simulation of a system consisting of 100 users and where 30% are standard users, 17% late night users, 50% email checkers and 3% system users. The lines at the bottom are the standard deviation of the data. For both the simulation and the real world data, each point on the curve is an average over data taken each minute over an interval of 30 minutes.

Suppose the average value for the number of users at a given time of the day is 19.9 and that the standard deviation is 5.0 for the data collected so far in the simulation. If a new measurement of 30 users is made at this time it deviates by 10.1 from the average value and this equals 2.02 standard deviations, since $2.02 \times 5.0 = 10.1$. This data point will then fall into the standard deviation slot ranging from 2.01 to 2.04. The deviation is scaled in this way in order to characterize the complete set of data at the same time.

The distribution of deviations seems to have a skewness to the left with a tail to the right when the measured value is close to zero some time of the week, for instance at night as shown for the dotted distribution of Figure 8. The only difference when performing this simulation compared to simulation leading to the data of the fully drawn line which show a symmetric Gaussian distribution of data points, is that the 10% of the users stay logged on at night in the latter simulation.

The skewness may be due to all the data points which are zero and therefore occurs slightly below the average. The non-zero points in the same area will be several standard deviations above the mean and therefore contribute to the tail of the skew contribution. When the value considered never is close to zero, the distribution is Gaussian as one would expect.

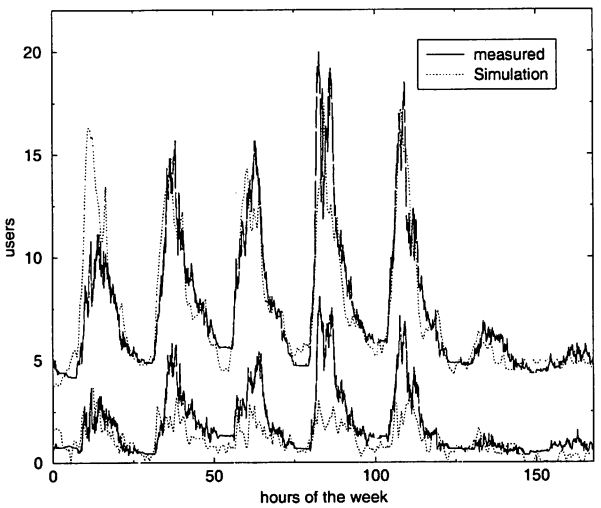


Figure 6: The number of users logged on as function of time. The fully drawn line is the average number of users during a week at a computer at Oslo University College averaged over four weeks. The dotted line are from a four week long simulation of a system consisting of 150 users and where 31% are standard users, 5% starting early, 2% late night users, 2% night users, 56% email checkers and 4% system users. The lines at the bottom are standard deviation of the data. Each point on the curves is an average over data taken each minute over an interval of 5 minutes.

On the other hand, when the value measured is limited from above, as the number of processes is at a given host, one would expect that the distribution would be skew to the right with a tail to the left, since there will be a lot of points close to the upper limit contributing to data points shortly above the average. This means that the skewness of the distribution, which is a well defined statistical quantity, might be used as a numeric measure of how limited a resource is. Frequently occurring values close to zero makes it skew to the left, while frequently occurring values close to the upper limit makes it skew to the right.

Some work remains to be done in order to understand this behavior thoroughly and quantitatively. Other characteristics will surely emerge as further studies are made. Such characteristics can be used to plan resources for future refinements of performance and reliability.

Conclusion

Although the first results of our simulations are promising, we have only scratched the surface of user-

level Monte Carlo simulation of computer systems. There are many possible new uses of such a general approach to investigating a computer system. One of

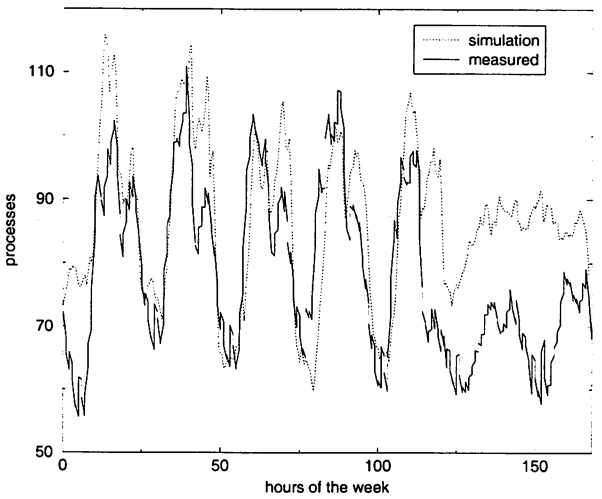


Figure 7: The number of processes as function of time. The fully drawn line is the average number of processes during a week at computer at Kansas State University averaged over nine weeks. The dotted line are from nine-week simulation of a system consisting of 150 users. User-data from the same computer and the same simulation where presented in Figure 5.

them is to study the effect of introducing the topography of the system, including network connections between the various hosts and servers. This will make it possible to simulate the network traffic and to find the solutions which gives the best and most balanced use of the resources of the system as a whole. As Traugott and Huddleston have pointed out, modern distributed systems are best thought of as a single virtual machine, at the level of the network [21]. In version 2 of cfengine [22], being developed at Oslo University College, the results of these studies are already being used to implement automatic anomaly detection for automatic regulation of systems. We hope to return to more intricate studies in future work.

The authors are grateful to Mark Burgess for inspiring discussions and to Tim Bower for providing us with data from Kansas State University. The simulation software can be obtained by contacting the authors.

References

[1] Burgess, M., "Theoretical system administration." *Proceedings of the Fourteenth Systems Administration Conference (LISA XIV)*, USENIX Association, Berkeley, CA, p. 1, 2000.

[2] Burgess, M., *Principles of Network and System Administration*, J. Wiley & Sons, Chichester, 2000.

[3] Burgess, M., H. Haugerud, T. Reitan, and S. Straumsnes, "Measuring Host Normality," *ACM/Transactions on Computing Systems*, submitted, 2001.

[4] Burgess, M., "Thermal, Non-Equilibrium Phase Space for Networked Computers," *Physical Review E*, Vol. 62, p. 1738, 2000.

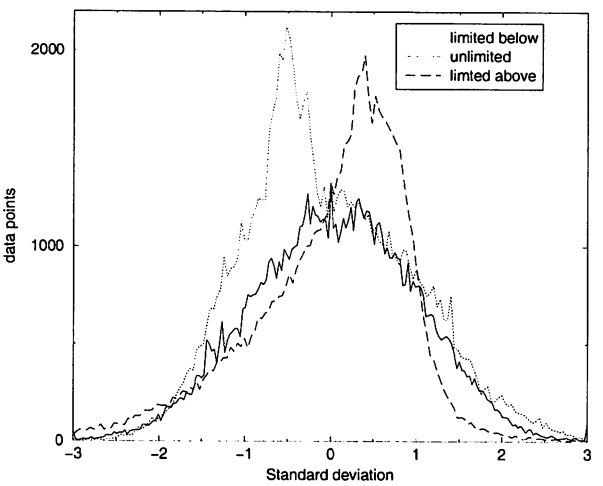


Figure 8: The fully drawn and the dotted line is the distribution of the deviation from the mean number of users logged on. The deviation of the x-axis is measured in number of standard deviations. The distribution which is skew to the left is from a simulation containing no or few users at night, yielding a lot of measurements close to zero, while the symmetric Gaussian distribution is from a simulation where the number of users rarely is zero. The dashed line is deviation from the mean number of processes for a simulation where many measurements are close to the upper limit of 256 processes.

[5] Burgess, M., "The Kinematics of Distributed Computer Transactions," *International Journal of Modern Physics*, Vol. C12, pp. 759-789, 2000.

[6] Ousterhout, J. K., H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson, "A Trace-driven Analysis of the Unix 4.2 BSD File System," *ACM/SOSP*, p. 15, 1985.

[7] Baker, M. G., J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout, "Measurements of a Distributed File System," *ACM/SOSP*, October, p. 198, 1991.

[8] Park, A., and J. C. Becker, "Measurements of the Paging Behavior of Unix," *Performance Evaluation Review*, Vol. 19, p. 216, 1991.

[9] Hellerstein, J. L., F. Zhang, and P. Shahabuddin, "An Approach to Predictive Detection for Service Management," *Proceedings of IFIP/IEEE INM VI*, p. 309, 1999.

[10] Chen, J. Cradley, Y. Endo, D. Mazieres, A. Dias, M. Seltzer, and M. D. Smith, "The Measured Performance of Personal Computer Operating Systems," *ACM Transactions on Computing Systems and Proceedings of the Fifteenth ACM*

symposium on Operating System Principles, 1995.

[11] Smith, A. J., "Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms," *IEEE transactions on Software Engineering*, Vol. SE-7, p. 403, 1981.

[12] Leland, W. E., and T. J. Ott, "Load Balancing Heuristics and Process Behavior," *Performance Evaluation Review*, Vol. 14, 54, 1986.

[13] Harchol-Balter, M., and A. B. Downey, "Exploiting Process Lifetime Distributions for Dynamic Load Balancing," *Performance Evaluation Review*, Vol. 24, 1996.

[14] Borst, S. C., "Optimal Probabilistic Allocation of Customer Types to Servers," *Performance Evaluation Review*, Vol. 23, p. 116, 1995.

[15] Crovella, M. E., and A. Bestavros, "Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes," *Performance Evaluation Review*, Vol. 24, p. 160, 1996.

[16] Willinger, W., V. Paxson, and M. S. Taqqu, "Self-Similarity and Heavy Tails: Structural Modelling of Network Traffic," in *A practical Guide to Heavy Tails: Statistical Techniques and Applications*, pp. 27-53, 1996.

[17] Fuchs, E., and P. E. Jackson, "Estimates of Distributions of Random Variables for Certain Computer Communications Traffic Models," *Communications of the ACM*, Vol. 13, p. 752, 1970.

[18] Berry, R. F., and J. L. Hellerstein, "Characterizing and Interpreting Periodic Behavior in Computer Systems," *Performance Evaluation Review*, Vol. 20, p.241, 1992.

[19] Glance, N., T. Hogg, and B. A. Huberman, "Computational Ecosystems in a Changing Environment," *International Journal of Modern Physics*, Vol. C2, p. 735, 1991.

[20] Law, Averill M., and W. David Kelton, *Simulation Modeling and Analysis*, McGraw-Hill, Boston, 2000.

[21] Traugott, S., and J. Huddleston, "Bootstrapping an Infrastructure," *Proceedings of the Twelth Systems Administration Conference (LISA XII)*, USENIX Association, Berkeley, CA, p. 181, 1998.

[22] Burgess, M., "A Site Configuration Engine," *Computing Systems*, MIT Press, Cambridge, MA, Vol. 8, p. 309, 1995.

Specific Simple Network Management Tools

Jürgen Schönwälder – Technical University of Braunschweig

ABSTRACT

The Simple Network Management Protocol (SNMP) has been around for more than a decade and is supported by most network devices and end systems. Despite this success, there is still a lack of simple to use network management applications.

This paper describes the design of an SNMP management tool called `scli` which provides an easy and efficient to use SNMP command line interface. The software architecture has been designed to make it easy for C programmers without any special SNMP programming skills to extend the functionality provided by `scli`.

Introduction

The Simple Network Management Protocol (SNMP) [1, 2] has been around for more than a decade and is supported by most network devices and end systems. Despite its success in the network devices, there is still a lack of simple to use network management applications.

High-end management platforms such as HP's OpenView [3] or Aprisma's Spectrum [4] are relatively expensive and require special training in order to use them effectively. Especially operators of smaller enterprise networks often cannot afford the purchase and training costs associated with these high-end management platforms. So they often revert to low-level SNMP tools such as `snmpwalk` in combination with shell scripting languages to get a certain job done quickly. Others use SNMP extensions of well known scripting languages such as Perl or Tcl to implement their own little management solution. While this seems to be a good approach in the short-term, these scripts tend to be loaded with site-specific details which often prevents people from sharing them. Furthermore, these scripts are often fragile and hard to maintain since error handling is usually poor and documentation is often missing.

This paper describes the design and implementation of an SNMP management tool called `scli` which provides an efficient to use command line interface to display, modify and monitor data retrieved from SNMP agents. It runs on simple ASCII terminals and does not require any graphical user interface capabilities. The tool provides command line editing, completion and history capabilities to make it easy for network operators and system administrators to use this tool, even if they cannot remember the precise command syntax.

The `scli` commands are organized in a logical command tree and hide the details of the SNMP interactions and the underlying MIB data structures. The default `scli` output format is optimized for human readability and in some cases resembles the output produced by existing Unix commands.

Optimizing the default output format for human readability has the disadvantage that it becomes harder to use `scli` in scripts since parsing the output is complicated and error prone. In order to use `scli` as a mechanism to collect data to be stored in data bases or displayed on web pages, a second XML-based output format has been implemented for many of the `scli` commands.

The `scli` software architecture has been designed to allow C programmers without knowledge of low-level SNMP APIs to extend the functionality provided by `scli`. This hopefully encourages a larger group of people to write and share extensions for specific device types, protocols or managed services.

This paper is organized as follows. The first section discusses the difference between generic and specific SNMP tools and why there is a need of specific rather than generic tools. The next section describes the overall software design of the simple command line management tool `scli`. The following section discusses some of the `scli` commands and presents examples how they can be used in practice. Subsequent sections describe the implementation of `scli` and how it can be extended. The paper concludes with some remarks on future work.

Generic vs. Specific Tools

The tool described in this paper was written because of the author's continued frustration how complex and inconvenient it is to configure, troubleshoot and monitor SNMP manageable devices. An analysis of the tools available shows that most of them fall into one of the following five categories:

Generic low-level SNMP Tools. The first category includes simple low-level SNMP command-line tools such as `snmpwalk` or `snmpset`. These tools are low-level since they just provide command-line interfaces to perform a

single or a sequence of related SNMP protocol operations on MIB variables. The tools generally do not understand the semantics of the data they manipulate. Furthermore, they require a certain amount of SNMP and MIB knowledge to interpret the results correctly.

Generic low-level SNMP APIs. The second category includes tools and libraries that give programmers a relatively low-level programmatic interface to invoke SNMP protocol operations and to access MIB definitions. Examples of generic low-level scripting APIs are WinSNMP [5], SNMP++ [6], NET-SNMP or the Tnm extension for Tcl [7, 8]. Many network operators and system administrators seem to prefer APIs that are based on scripting languages such as Perl or Tcl over APIs for system programming languages such as C or C++ since they are much easier to deal with.

Generic MIB Browsers. The third category includes programs that allow network operators to browse through MIB data on SNMP-enabled devices. Some MIB browsers use Web technologies for their user interface while the majority provides graphical user interfaces.
MIB browsers are usually designed as generic tools that do not really understand the semantics of the data they display and manipulate. Many browser are able to load and interpret MIB module definitions at run-time and some of the more advanced browsers allow users to customize the displays to a large extent. However, many important semantics described in MIB description clauses are not machine readable. Therefore, generic MIB browsers generally require that users are familiar with the semantics of MIB variables or at least able to read and understand MIB module definitions.

Generic Monitoring Tools. The fourth category includes generic monitoring tools. MRTG [9] is an example of a generic monitoring tool which is used in many networks to gather statistics and to detect unusual system behavior. Some of these tools have limitations that cause them to produce erroneous results in some situations (handling of counter discontinuities) if the user configuring these tools does not pay attention to special MIB semantics. Some more specific tools such as Cricket [10] have been implemented on top of these generic monitoring tools to simplify the configuration for typical use cases.

Generic Management Platforms. The fifth category consists of management platforms which provide a generic infrastructure for the implementation of network management applications. Applications written on top of these platforms make use of platform specific interfaces and services, such as protocol APIs or database services. Platforms also often include generic tools for monitoring, event correlation or topology discovery.
The `tkined` [11] and `gxsnmp` packages are examples of openly available platforms. More complex examples are commercial management platforms such as HP's OpenView [3] or Aprisma's Spectrum [4].

There are many SNMP management tools available today which fall into one of the five categories described above. But the author still often feels uncomfortable when trying to use them, even though he has implemented some of these generic tools himself in the past. MIB browsers which display raw MIB data structures tend to be of little use for actual management because MIB data structures are usually designed to be read by programs rather than humans. Furthermore, MIB modules undergo revisions as the networking technologies evolve. Sometimes, the original MIB module design turns out to be problematic and the attempts to maintain backwards compatibility while supporting new features makes MIB module data structures often hard to understand.

Another problem are the naming schemes used in MIB modules, which are typically optimized for machines and which do not necessarily reflect what humans prefer to use. For example, humans prefer to identify network interfaces by names, such as "eth0". MIB modules, however, use numbers to identify network interfaces [12]. Furthermore, on some devices, these numbers can change with every re-initialization. A good specific tool should allow users to refer to interfaces by name and it should hide the SNMP and MIB specific naming details.

Generic tools often do not understand the relationships between MIB objects. For example, consider the speed of a network interface. There are two standardized objects in the IF-MIB [12] that report the speed of a network interface (`ifSpeed` and `ifHighSpeed`) and a good application should understand their semantics and relationship and display the correct value in a meaningful way. Generic MIB browsers are not able to do that and put the burden on the user to understand how `ifSpeed` and `ifHighSpeed` relate to each other and to pick the right value.

It seems that the approach to build generic tools that expose raw MIB data structures and which require that the network operator or system administrator has SNMP and MIB knowledge does not work very well in practice. There is a need for simple tools that are specific instead of generic in the sense that they understand the semantics of the data they manipulate and hide low-level SNMP and MIB details. Specific tools should be designed to do just one thing and they should attempt to do it right. Specific management tools must be written by programmers who do understand the semantics of the MIB objects as well as the

conceptual model behind the relevant MIB modules [13].

The `scli` tool, which was born in January 2001 and which has been openly available since February 2001, has since its first release improved in many directions and it shows that the implementation of efficient specific network management tools such as `scli` can be easy and fun if one chooses a suitable software design.

Software Design

This section discusses the software design behind `scli`. The software design addresses five key requirements:

1. The first requirement is extensibility. The software design should make it relatively easy to add new features to `scli`. This requires that the internal APIs are as simple as possible. Furthermore, the code must be obvious so that people can easily derive extensions from the existing code base. In order to get many programmers involved, it is necessary to hide low-level SNMP communication details as much as possible.
2. The second requirement is robustness. The software design should ensure as much as possible that errors are detected and handled gracefully where possible. This implies to use facilities which help to avoid problems such as buffer overruns and to validate data received from the network before processing it. Furthermore, the program should abort if coders forget to check for possible error conditions as soon as possible so that bugs are noticed and fixed.
3. The third requirement is maintainability. It must be possible to evolve the software over time, which includes internal API changes. Furthermore, it is important to ensure that the documentation is available and in sync with the implementation since `scli` users are not expected to read MIB modules in order to use `scli`. Finally, it is important to stay focussed in the overall scope so that the resources available can be used effectively.
4. The fourth requirement is efficiency. The implementation should be efficient regarding the amount of resources needed to implement a given management operation. This is of special importance if the tool is used in scripts that perform more complex management tasks. Some of the well-known generic low-level SNMP tools consume noticeable resources while parsing MIB files during startup and are thus inefficient if they are called frequently in scripts.
5. The fifth requirement is portability (at least across Unix platforms). A port to Win32 platforms should be possible, although the author does not really have a need for such a port himself.

Implementation Language

The author's experience with the Tnm extension for Tcl, a generic low-level SNMP API [7], shows that only a few Tcl coders actually contribute scripts written on top of the Tnm API back to the package maintainer. And those who do so usually do not care too much about the overall code organization and the scripts often depend on side specific details. While the original motivation behind Tnm was to provide a solid SNMP scripting API which should make it easy for people to create a repository of useful management scripts, the overall success in reaching that goal is rather limited. In fact, many management scripts turn out to be rather fragile and trying to maintain them is relatively costly.

It is interesting to note that the author's experience with other open source projects that are coded in C is quite different. Contributed patches for C code are often of good quality and much easier to integrate and maintain. Furthermore, compiled languages greatly help to detect many potential problems and inconsistencies if internal APIs change. It was therefore decided to implement `scli` entirely in C.

C++ was also considered as a potential implementation language but was finally rejected since the benefits of C++ over C relative to the requirements stated above are limited and C is still more portable and efficient and the number of C programmers is still bigger than the number of C++ programmers. The Java language was considered but not selected since the resource consumption is noticeable.

Software Architecture

The overall software architecture is shown in Figure 1. The package uses the `glib` library to achieve portability and to reuse generic data structures such as lists and dynamic strings. The SNMP engine `gsnmp` has been derived from the `gxsnmp` package and was subsequently modified to fix bugs and to improve stability. The SNMP engine itself uses `glib`.

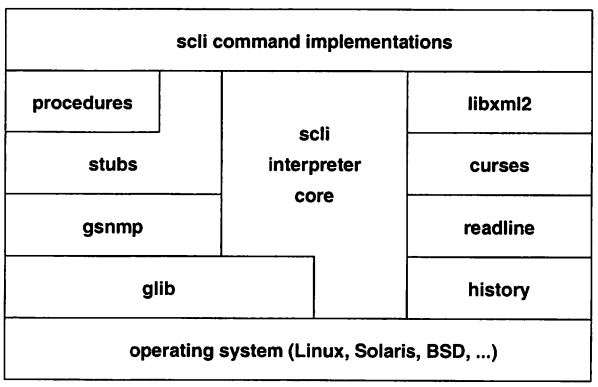


Figure 1: Software architecture.

The `gsnmp` library provides roughly the same low-level functionality as many other SNMP APIs. Since it was felt necessary to hide programmers from

the low-level SNMP programming details, it was decided to use a MIB compiler to generate C stubs from MIB modules. The stubs consist primarily of C structures which represent MIB table rows or groups of scalars plus a set of stub functions which can be used to read/write these structures. The implementations of the stub functions serialize/deserialize the C structures into SNMP varbind lists. They also validate the data to ensure that the elements of the varbind lists have appropriate types and sizes. The stub code generator is further described below.

The `scli` command implementations either use the stubs directly or they use so called MIB procedures. MIB procedures extend the stub interface with specific functions for common operations like creating rows in certain MIB tables or iterating over certain MIB tables. MIB procedures are by definition MIB specific and implemented entirely by using the stub interface.

The `scli` interpreter core provides all the infrastructure needed to register commands, to tokenize the input stream, to locate and execute the function implementing a recognized command and to finally display the results on stdout or via a pager. The interpreter uses the GNU `readline` and `history` libraries for command line editing and the `curses` library for screen management. It also uses `glib` data types internally. All state information is bound to the `scli` interpreter. It is thus possible to have multiple `scli` interpreters in a single process – although this feature is currently not used.

The design decision to implement our own interpreter instead of using one of the available embeddable command interpreters was driven by the observation that the features needed by `scli` are very small and most interpreters such as Tcl are too heavy weight these days for simple tools like `scli`.

The interpreter core and some command implementations also use the `libxml2` library to create and manipulate XML documents. By using a dedicated XML library, it is possible to ensure that any generated XML output is well-formed.

User's View

This section describes `scli` from a user's point of view. It briefly introduces the basics about the structure of `scli` commands before presenting some examples how `scli` can be used.

Command Overview

All `scli` commands are hierarchically structured with a small set of top-level commands. The first five top-level commands (`open`, `close`, `exit`, `help`, `history`) are used to open and close SNMP sessions and to help with user interactions. The remaining top-level commands can be used to create/delete something (`create`, `delete`), to modify something (`set`), to display or monitor data (`show`, `monitor`), or to produce an `scli` script which restores the current configuration (`dump`). As an example, Figure 2 shows part of the `set` command hierarchy.

```
- set
   |- system
   |   |- contact
   |   |- name
   |   `- location
   |- ip
   |   |- forwarding
   |   `- ttl
   `- interface
       |- status
       |- alias
       |- notifications
       `- promiscuous
```

Figure 2: `scli set` command hierarchy.

Commands are also logically organized into `scli` modes. Figure 3 shows the syntax of the commands provided by the `interface` mode. Note that interfaces can be selected by regular expressions that are matched against the interface description. This allows users to perform a single operation on a set of interfaces.

```
set interface status <regexp> <status>
set interface alias <regexp> <string>
set interface notifications <regexp> <value>
set interface promiscuous <regexp> <bool>
show interface info [<regexp>]
show interface details [<regexp>]
show interface stack [<regexp>]
show interface stats [<regexp>]
monitor interface stats [<regexp>]
dump interface
```

Figure 3: `scli` interface mode.

`scli` supports the concept of recursive command evaluation, which is especially useful for the `show` and `dump` commands. The `show interface` command will retrieve and display all information about all network interfaces while the `show` command will retrieve and display all information available from a device.

Interactive Browsing and Monitoring

The `show` and `monitor` commands can be used to interactively inspect and monitor a device. The screenshot in Figure 4 shows how `scli` displays the containment hierarchy of the physical entities that make up a router. Additional `scli` commands can be used to get more detailed information about the physical entities.

Sometimes it is necessary to quickly monitor some core statistics in order to track down a network problem. The `monitor` commands can be used for this purpose. Figure 5 shows a screenshot where `scli` shows basic statistics and status information for the network interfaces of a router. Note how `scli` handles data not available for some ATM layers.

The look & feel of the monitoring commands is similar to the well-known Unix `top` command. The top half of the screen displays basic summary information. Special keys can be used to customize the monitoring display.

Configuring Virtual LANs

LAN bridges (sometimes called layer 2 switches) can often be configured via SNMP. The author recently had a need to configure virtual LANs across a number of bridges. The telnet interface provided by the bridges is menu-driven and not easy to handle for automated configuration. However, the bridges support vendor specific MIB objects to allow configuration via SNMP. (Unfortunately, the devices do not support the standard MIB for virtual LANs as defined in RFC 2674 [14].)

The approach to solve the configuration problem was to extend `scli` with commands that can create/delete virtual LANs and commands to assign ports to them. This allows to save the virtual LAN configuration for each bridge in a simple ASCII file. By using the `m4` macro processor, it is easy to import shared bridge configuration commands and to use symbolic names for port sets. Figure 6 shows the `scli` script to install the virtual LANs. Note the regular expression at the beginning to first remove all relevant virtual LANs. Figure 7 shows the `scli` script which configures the ports on a particular bridge.

```
local
Agent Boot Time:  2001-09-24 17:21:51 +02:00
Interfaces:       12
Bridge Type:      source route transparent (SRT)
(ciscobs.rz) scli > show entity containment
ENTITY CLASS     CONTAINMENT
     1 chassis     7206VXR chassis, Hw Serial#: 21275454, Hw Revision: D
     2 module      |- NPE 300 Card, Hw Serial#: 21275454, Hw Revision: D
     3 container   |- Chassis Slot
     4 module      |  `- I/O FastEthernet (TX-ISL)
     5 port        |     `- DEC21140A
     6 container   |- Chassis Slot
     7 module      |  `- 2 Port Fast Ethernet/ISL 100BaseTX Port Adapter
     8 port        |     |- AmdFE
     9 port        |     `- AmdFE
    10 container   |- Chassis Slot
    11 module      |  `- POS Port Adapter (SM)
    12 port        |     `- Packet over Sonet
    13 container   |- Chassis Slot
    14 container   |- Chassis Slot
    15 module      |  `- ATM Lite Port Adaptor (SM)
    16 port        |     `- TI1570 ATM
    17 container   |- Chassis Slot
    18 container   `- Chassis Slot
(ciscobs.rz) scli >
```

Figure 4: `scli` showing the containment structure of router components.

```
local
Agent:    ciscobs.rz:161 up 9 days 23:34:35                          15:56:26
Descr:    Cisco Internetwork Operating System Software   IOS (tm) 7200 Software
IPv4:     6435 pps in 6408 pps out 6399 pps fwd    0 pps rasm    0 pps frag
UDP:         8 pps in    6 pps out
TCP:         0 sps in    0 sps out    0 con est    0 con aopn    0 con popn
Command:  monitor interface stats

INTERFACE STATUS I-BPS O-BPS I-PPS O-PPS I-ERR O-ERR  DESCRIPTION
        1  UUCN     1m    2m  3270  3152     0     0  FastEthernet0/0
        2  UUCN      0    23     0     0     0     0  FastEthernet1/0
        3  UUCN    10k   10k    50    50     0     0  FastEthernet1/1
        4  UUCN     2m    1m  3197  3254     0     0  POS2/0
        5  UDCN      0     0     0     0     0     0  ATM4/0
        6  UD--   ----  ----  ----  ----  ----  ----  ATM4/0-atm layer
        7  UD--   ----  ----  ----  ----  ----  ----  ATM4/0.0-atm subif
        8  UDNN      0  ----     0  ----  ----  ----  ATM4/0-aal5 layer
        9  UDNN      0  ----     0  ----  ----  ----  ATM4/0.0-aal5 layer
       10  UUNN      0  2105     0    18     0     0  Null0
       11  UUNN      0     0     0     0     0     0  Loopback0
       12  UUNN      0     0     0     0     0     0  Tunnel34
```

Figure 5: `scli` monitoring network interface statistics.

```
# cleanup - regexps are cool :-)
delete nortel bridge vlan "^(134|ibr-)"

# IBR vlans (544-559)   134.169.34.*
create nortel bridge vlan 544 ibr-core
create nortel bridge vlan 545 ibr-cip
create nortel bridge vlan 546 ibr-test
create nortel bridge vlan 547 ibr-wlan
```

Figure 6: `scli` script for creating virtual LANs.

```
define(UP,'25,185')      # uplink ports
define(WLAN,'2,56')      # wireless vlan
define(CORE,'1,3-24,33-55,65-88')
                         # core vlan

# create the vlans:

include(vlan-all.scli)

# vlan port assignments:

set nortel bridge vlan ports \
                 ibr-core UP,CORE
set nortel bridge vlan default \
                 ibr-core CORE

set nortel bridge vlan ports \
                 ibr-wlan UP,WLAN
set nortel bridge vlan default \
                 ibr-wlan UP,WLAN
```

Figure 7: `scli` script which creates virtual LANs and assigns ports.

It is also useful to be able to dump the virtual LAN configuration via the `dump` command from the device in order to check whether it matches the configuration that is supposed to be on the device.

Generating HTML Status Pages

It is often convenient to generate HTML status pages for some devices (such as printers) which are linked to the Intranet. These status pages allow users to figure out why for example a print job does not progress in the print queue by looking at a virtual printer console. The `Printer-MIB` [15], which is supported by many printers, provides a simple way to read the console display and the status of the printer lights.

Generating HTML status pages is straight-forward since `scli` can generate XML output. XSL transformations can turn the `scli` XML output into a nice HTML page. Figure 8 shows the core of a transformation which shows console lights as an HTML table.

Programmer's View

This section introduces `scli` from a programmer's point of view. It first describes the stub code generator before explaining how a simple `scli` command to display the console lights of printers is implemented and registered.

Stub Generator

The stub code generator is a key component since it hides the low-level SNMP communication details. The stub code generator takes a MIB module as input and generates a pair of .h and .c files for the MIB module. The header file contains C type definitions for MIB table rows or groups of scalars. The SMI data types are mapped to `glib` data types according to the base data type model used by the SMIng proposal [16]. Some additional structure members whose names start with an underscore are introduced when dealing with variable size MIB variables.

The members of the generated C structures are usually pointers. This reflects the fact that SNMP agents are not required to return values for all variables, either due to implementation limitations or due

```
<xsl:template match="console">
  <table>
    <tr>
      <xsl:for-each select="light">
        <xsl:element name="td">
          <xsl:attribute name="width">60</xsl:attribute>
          <xsl:attribute name="align">center</xsl:attribute>
          <xsl:if test="status != 'off'">
            <xsl:attribute name="bgcolor">
              <xsl:value-of select="color"/>
            </xsl:attribute>
          </xsl:if>
          <xsl:if test="status = 'blink'">
            <xsl:element name="span">
              <xsl:attribute name="style">text-decoration:blink</xsl:attribute>
            </xsl:element>
          </xsl:if>
          <xsl:apply-templates select="description"/>
        </xsl:element>
      </xsl:for-each>
    </tr>
  </table>
</xsl:template>
```

Figure 8: XSL transformation for `show printer console` XML output.

to access control. Variables that are not accessible will be represented by a NULL pointer. The decision to use pointers requires that programmers check carefully whether the pointers are valid before using them. Failure to do so will result in segmentation faults – a clear indication that the program is buggy and must be fixed. An alternative option would have been to introduce special bit fields which indicate whether a given data member is valid or not. This option was rejected since programmers will likely forget to check these bit fields and programs will operate on invalid data without being noticed.

The generated header file also defines several stub functions that read/write the C structures from/to SNMP agents. Stub functions that retrieve complete MIB tables return the data to the application as an array of pointers to the C structures representing table rows. The read/write stub functions also have a mask argument which can be used to specify that only a subset of the members of the C structure should be read/written.

The implementation of the stub functions is contained in the .c files. The table retrieval stubs generate a sequence of suitable SNMP requests to read a table. Holes in tables are handled automatically and data which can be obtained by unpacking instance identifiers is not retrieved explicitly in order to save some bandwidth.

The data contained in response messages is first validated by doing some basic type and range/size checking. The instance identifier is unpacked and validated before C structures are filled with appropriate values. Detected errors are signaled using `glib` warnings and the values are ignored. This ensures that SNMP communication problems are noticed and that applications using the stubs only operate on validated values.

The stubs also provide mapping tables for enumerations. These tables can be used to map numbers or object identifier values to labels and vice versa. However, in many cases, the labels assigned in MIB modules are rather useless for direct display because they are either too cryptic or simply too long. It is thus not uncommon to implement more specific mapping tables in addition to the tables generated by the MIB compiler.

Figure 9 shows the stubs that are generated for the `prtConsoleLightEntry` of the Printer-MIB

```
/*
 * C type definitions for Printer-MIB::prtConsoleLightEntry.
 */

typedef struct {
    gint32   hrDeviceIndex;
    gint32   prtConsoleLightIndex;
    gint32   *prtConsoleOnTime;
    gint32   *prtConsoleOffTime;
    gint32   *prtConsoleColor;
    guchar   *prtConsoleDescription;
    gsize    _prtConsoleDescriptionLength;
} printer_mib_prtConsoleLightEntry_t;

extern void
printer_mib_get_prtConsoleLightTable(GSnmpSession *s,
        printer_mib_prtConsoleLightEntry_t ***prtConsoleLightEntry,
        gint mask);

extern void
printer_mib_free_prtConsoleLightTable(
        printer_mib_prtConsoleLightEntry_t **prtConsoleLightEntry);

extern printer_mib_prtConsoleLightEntry_t *
printer_mib_new_prtConsoleLightEntry(void);

extern void
printer_mib_get_prtConsoleLightEntry(GSnmpSession *s,
        printer_mib_prtConsoleLightEntry_t **prtConsoleLightEntry,
        gint32 hrDeviceIndex, gint32 prtConsoleLightIndex, gint mask);

extern void
printer_mib_set_prtConsoleLightEntry(GSnmpSession *s,
        printer_mib_prtConsoleLightEntry_t *prtConsoleLightEntry,
        gint mask);

extern void
printer_mib_free_prtConsoleLightEntry(
        printer_mib_prtConsoleLightEntry_t *prtConsoleLightEntry);
```

Figure 9: Stub interface for `prtConsoleLightEntry` of the `Printer-MIB` (RFC 1759).

[15] which describes the status of a printer console light. All type and function declarations are prefixed by the MIB module name in order to deal with potential name clashes. The first two stub functions operate on complete tables while the remaining stub functions operate on table rows. The stub code generator has been implemented as an output driver of the `smidump` MIB compiler.

Command Implementation

Figure 10 shows the implementation of the `show printer console lights` command. Commands are implemented as C functions which are called with a handle for the `scli` interpreter and the command arguments as input and return an `scli` return code. The commands usually first check the command arguments before retrieving the data they manipulate. If the retrieval was successful, they start manipulating the data. The last section of a command implementation is responsible to free any allocated resources.

The `show_printer_console_lights()` function shown in Figure 10 first iterates over the retrieved table to calculate the maximum length of the description strings. The second iteration calls an output formatting function for each table row, depending on the current state of the `scli` interpreter. The result is written into the interpreter, which provides either `glib` dynamic strings or a suitable pointer to an `libxml2` node.

The default output formatting function for the `show printer console lights` command is shown in Figure 11. It uses a utility function `fmt_enum()` to lookup the label for a color number and it does some computations to figure out whether the light is on, off or blinking.

Command Registration

Command implementations must be registered in the `scli` interpreter as shown in Figure 12. This is accomplished by creating an array of command descriptions. Each command description contains the command name, the description of the arguments accepted by the command, the documentation of the command and some command flags. Commands that are able to produce XML output also describe the XML path and the XML Schema definition (not shown in Figure 12).

Commands always belong to an `scli` mode. The structure which describes an `scli` mode contains the name of the mode, the documentation of the mode and the commands it provides.

```
static int
show_printer_console_lights(scli_interp_t *interp, int argc, char **argv)
{
    printer_mib_prtConsoleLightEntry_t **lightTable;
    int i;
    int light_width = 12;

    if (argc > 1) return SCLI_SYNTAX;

    printer_mib_get_prtConsoleLightTable(interp->peer, &lightTable, 0);
    if (interp->peer->error_status) return SCLI_SNMP;

    if (lightTable) {
        for (i = 0; lightTable[i]; i++) {
            if (lightTable[i]->_prtConsoleDescriptionLength > light_width) {
                light_width = lightTable[i]->_prtConsoleDescriptionLength;
            }
        }
        if (! scli_interp_xml(interp)) {
            g_string_sprintfa(interp->header,
                              "PRINTER LIGHT %-*s STATUS COLOR",
                              light_width, "DESCRIPTION");
        }
        for (i = 0; lightTable[i]; i++) {
            if (scli_interp_xml(interp)) {
                xml_printer_console_light(interp->xml_node, lightTable[i]);
            } else {
                fmt_printer_console_light(interp->result, lightTable[i],
                                          light_width);
            }
        }
    }

    if (lightTable) printer_mib_free_prtConsoleLightTable(lightTable);

    return SCLI_OK;
}
```

Figure 10: Implementation of the `show printer console lights` command.

All command and mode documentation is defined in the C code and registered within the interpreter. This encourages programmers to provide documentation when implementing new commands. The `scli` manual page and other documentation is generated automatically from the output produced by the `show scli modes` and the `show scli schema` commands.

The printer mode registration shown in Figure 12 registers the function `show_printer_console_lights()` twice. The second registration causes the command to be executed periodically since it sets the `SCLI_CMD_FLAG_MONITOR` flag.

Conclusions

This paper first motivated the need for specific rather than generic SNMP-based management tools. It then presented the overall software design for a specific management tool called `scli` before discussing `scli` in some more depth from the user's and the programmer's point of view.

The evolution of `scli` so far has shown that the approach of using compiler generated stubs to hide low-level SNMP communication details is feasible. Furthermore, the number of commands available is already large enough to inspect, monitor and configure devices using `scli`.

There are of course a number of areas where additional work can be done. The SNMP engine does not yet support SNMPv3 security [2]. SNMPv3 is slowly getting more widespread deployment and it would be nice to take advantage of strong security, especially for `set` and `create` commands.

The code generator can be improved in many ways. The biggest limitation right now is the restriction that stubs can only operate on table rows or groups of scalars. It is sometimes desirable to have atomic SNMP set operations on varbinds that include tabular data and scalars. The prominent example are spin-lock variables such as `snmpSetSerialNo` [17].

Many of the current MIB procedures follow similar patterns and it would be convenient to generate them automatically from formal MIB annotations. An annotation language would perhaps also enable us to automatically generate suitable caching strategies in order to reduce the amount of data retrieved from SNMP agents. However, some more experimentation is needed to better understand the requirements for such an annotation language.

Finally, it would be nice to have more command implementations. People who like the tool and its design are therefore encouraged to write new modes

```
static void
fmt_printer_console_light(GString *s,
                  printer_mib_prtConsoleLightEntry_t *lightEntry,
                          int light_width)
{
    const char *state = "off";
    const char *e;

    g_string_sprintfa(s, "%6d  ", lightEntry->hrDeviceIndex);
    g_string_sprintfa(s, "%4d  ", lightEntry->prtConsoleLightIndex);

    if (lightEntry->prtConsoleDescription) {
        g_string_sprintfa(s, "%-*.*s ", light_width,
                    (int) lightEntry->_prtConsoleDescriptionLength,
                          lightEntry->prtConsoleDescription);
    } else {
        g_string_sprintfa(s, "%*s", light_width, "");
    }

    if (*lightEntry->prtConsoleOnTime
        && !*lightEntry->prtConsoleOffTime) {
        state = "on";
    } else if (!*lightEntry->prtConsoleOnTime
               && *lightEntry->prtConsoleOffTime) {
        state = "off";
    } else if (*lightEntry->prtConsoleOnTime
               && *lightEntry->prtConsoleOffTime) {
        state = "blink";
    }
    g_string_sprintfa(s, " %-*s ", 5, state);

    e = fmt_enum(printer_mib_enums_prtConsoleColor,
                 lightEntry->prtConsoleColor);
    g_string_sprintfa(s, "%s\n", e ? e : "");
}
```

Figure 11: Implementation of the `show printer console lights` formatter.

for their favorite devices or protocols and to contribute them to the `scli` project.

Acknowledgments

The author likes to thank Frank Strauss for the many fruitful discussions which helped to shape the design of `scli` and his XSL transformations. The author is also grateful to the `scli` users who have contributed patches and who maintain packages for various Linux systems.

Availability

The SNMP command line interface `scli` has been released under the terms of the GNU General Public License version 2. The project home page is http://www.ibr.cs.tu-bs.de/projects/scli/. Debian and RPM packages for Linux systems have been contributed by `scli` users.

The stub code generator has been integrated into the `libsmi` package which has been released under Berkeley copyright conditions. The project home page is http://www.ibr.cs.tu-bs.de/projects/libsmi/.

Biography

Jürgen Schönwälder received his diploma in computer science in 1990 and his doctoral degree in 1996 from the Technical University of Braunschweig, Germany. His research interests are network management, distributed systems and network security. He has co-authored several network management related RFCs and is currently the chair of the Network Management Research Group (NMRG) of the Internet Research Task Force (IRTF).

References

[1] Stallings, W., *SNMP, SNMPv2, SNMPv3, and RMON1 and 2*, Addison-Wesley, Third edition, 1999.

[2] Case, J., R. Mundy, D. Partain, and B. Stewart, "Introduction to Version 3 of the Internet-standard Network Management Framework," *RFC 2570*, SNMP Research, TIS Labs at Network Associates, Ericsson, Cisco Systems, April, 1999.

[3] Blommers, J., *OpenView Network Node Manager: Designing and Implementing an Enterprise Solution*. Prentice Hall PTR, 2000.

[4] Lewis, L., *Managing Business and Service Networks*, Kluver Academic/Plenum Publishers, 2001.

[5] Natale, B., "WinSNMP v2.0 – Evolution of an Industry-standard API," *Simple Times*, Vol. 6, Num. 1, March, 1998.

[6] Mellquist, P. E., "SNMP++: An Object Oriented Approach to Network Management Programming," *Simple Times*, Vol. 7, Num. 1, March, 1999.

```
void
scli_init_printer_mode(scli_interp_t * interp)
{
    static scli_cmd_t cmds[] = {
        { "show printer console lights", NULL,
          "The show printer console lights command shows the current"
          "status of the printer's lights. [...]",
          SCLI_CMD_FLAG_NEED_PEER | SCLI_CMD_FLAG_XML,
          "printer console",
          "<xsd> <!-- ... --> </xsd>",
          show_printer_console_lights },
        { "monitor printer console lights", NULL,
          "The monitor printer console lights command shows the same"
          "information as the show printer console lights command. The"
          "information is updated periodically.",
          SCLI_CMD_FLAG_NEED_PEER | SCLI_CMD_FLAG_MONITOR,
          NULL, NULL,
          show_printer_console_lights },
        { NULL, NULL, NULL, 0, NULL, NULL, NULL }
    };
    static scli_mode_t printer_mode = {
        "printer",
        "The scli printer mode is based on the Printer-MIB as published"
        "in RFC 1759 and some updates currently being worked on in the"
        "IETF Printer MIB working group.",
        cmds
    };
    scli_register_mode(interp, &printer_mode);
}
```

Figure 12: Registration of the `show printer console lights` command.

[7] Schönwälder, J. and H. Langendörfer, "Tcl Extensions for Network Management Applications," *Proc. Third Tcl/Tk Workshop*, pp. 279-288, Toronto, July, 1995.

[8] Zeltserman, D. and G. Puoplo, *Building Network Management Tools with Tcl/Tk*, Prentice Hall, 1998.

[9] Oetiker, T., "MRTG – Multi Router Traffic Grapher," *Proc. Twelfth Conference on Large Installation System Administration (LISA XII)*, Boston, December, 1998.

[10] Allen, J. R., "Driving by the Rear-View Mirror: Managing a Network with Cricket," *Usenix First Conference on Network Administration*, April, 1999.

[11] Schönwälder, J. and H. Langendörfer, "How To Keep Track of Your Network Configuration," *Proc. Seventh Conference on Large Installation System Administration (LISA VII)*, pages 189-193, Monterey (California), November, 1993.

[12] McCloghrie, K. and F. Kastenholz, "The Interfaces Group MIB," *RFC 2863*, Cisco Systems, Argon Networks, June, 2000.

[13] Schönwälder, J. and A. Müller, "Reverse Engineering Internet MIBs," *Proc. Seventh IFIP/IEEE International Symposium on Integrated Network Management*, Seattle, May, 2001.

[14] Bell, E., A. Smith, P. Langille, A. Rijhsinghani, and K. McCloghrie, "Definitions of Managed Objects for Bridges with Traffic Classes, Multicast Filtering and Virtual LAN Extensions," *RFC 2674*, 3Com, Extreme Networks, Newbridge Networks, Cabletron Systems, Cisco Systems, August, 1999.

[15] Smith, R., F. Wright, T. Hastings, S. Zilles, and J. Gyllenskog, "Printer MIB," *RFC 1759,* Texas Instruments, Lexmark International, Xerox Corporation, Adobe Systems, Hewlett-Packard, March, 1995.

[16] Schönwälder, J. and F. Strauss, "Next Generation Structure of Management Information for the Internet," *Proc. Tenth IFIP/IEEE Workshop on Distributed Systems: Operations and Management*, pp. 93-106, Springer Verlag, October, 1999.

[17] Case, J., K. McCloghrie, M. Rose, and S. Waldbusser, "Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)," *RFC 1907*, SNMP Research, Cisco Systems, Dover Beach Consulting, International Network Services, January, 1996.

Gossips – System and Service Monitor

Victor Götsch, Albert Wuersch, and Tobias Oetiker – Swiss Federal Institute of Technology

ABSTRACT

Gossips is a modular client/server based system monitor. It uses distributed monitoring tasks to define and report states of an IT-environment. A monitoring task includes probes to measure data and a test to evaluate them. Gossips does not only report problems, it can also suggest solutions to the problems by consulting a knowledge-base, which is maintained and easily extended by the system managers using the local system. The monitor software is easily extensible through a flexible plug-in system for tests and probes. The monitor software is written in object oriented Perl which allows new tasks to inherit large parts of the existing infrastructure of the program.

Introduction

The problem of monitoring a group of networked hosts has been discussed at length only recently by John Sellens [1]. Many protocols and tools for monitoring are available, including SNMP [2], Big Brother [3], Swatch [4], Spong [5] and pikt [6]. These have different strengths and weaknesses. Our goal in this project was to address some of the problems we found with existing solutions, focusing on a clean architecture and easy extensibility. After an evaluation of the mentioned tools we defined the following criteria for a new design:

- The monitor relies on a scalable client-server architecture where the client only talks to the server when it finds a problem and periodically assures the server that everything is okay.
- The software design is flexible and expandable.
- Only free tools are used (e.g., GNU GPL).
- The monitoring system allows to archive solutions to known problems.

State of the Art

Evaluation Criteria

- **Configuration**: The tool configuration should present the system manager with a good overview of the system and services monitored. The configuration is the instrument of a system manager using the tool. The system manager should be able to change configurations quickly without editing many files.
- **Design and Complexity**: The design of the tool should be as simple as possible, but not too simple. This concerns not just the code, but also the documentation and configuration of the tool.
- **Scalability**: The tool should work fine with five as well as with 5000 machines.
- **Extensibility**: The tool should offer an interface for adding new monitoring tasks without modification to the code of the tool.
- **Modularity**: This is in fact a specific aspect of extensibility. When a new extension is added to the tool, it should be possible to reuse this new extension, like in a lego system.
- **Messaging**: The tool should report exceptions. It should not primarily display a webpage with red and green 'lights.' Such a webpage gives the system manager kind of a secure feeling when he sees all shining in green. But he always has to look at the pages and is distracted from his work.

Comparison

A first evaluation in summer 2000 showed deficiencies in most tools mainly in the areas of extensibility and modularity. The table in Figure 1 shows an updated summary of the evaluation based on the latest versions (September 2001) of the most promising monitor tools after our first evaluation.

Big Brother

- **Configuration**: Big Brother needs a separate configuration file for each local test. Global tests, such as network tests are all configured in one additional file. In these configuration files you are able to define global configuration for

Monitor	configuration	design	scalable	extensible	modular	messaging
Big Brother	o	+	+	o	-	+
Swatch	+	+	+	o	-	+
Spong	o	+	+	o	o	+
pikt	+	+	+	+	-	+
gossips	+	+	+	+	+	+

Figure 1: Comparison: '+' good; 'o' okay; '-' missing.

all client and local configuration for special clients. It is rather painful to make changes in this system of configuration files.

- **Extensibility**: There is an interface where you can add your own scripts. The script has to translate the state of your system to *green*, *yellow* and *red* states. This makes it difficult to write a test which looks for keywords in log-files. Additionally it is rather complicated to configure tests from the configuration file.
- **Modularity**: Since the new tests are hard to configure from the configuration file, it is difficult to write reusable Big Brother tests.

Swatch

- **Configuration**: It takes just a few minutes to learn how to setup and use the monitoring system. Each system manager has his customized swatch configuration file, that contains pattern/action pairs that are personally interesting, or that pertain to his system responsibility.
- **Extensibility**: Swatch is a monitoring tool to observe syslogs. It is possible to implement tests that write their monitored data to the syslog, but Swatch does not support a test developer with any tools to write a new test.

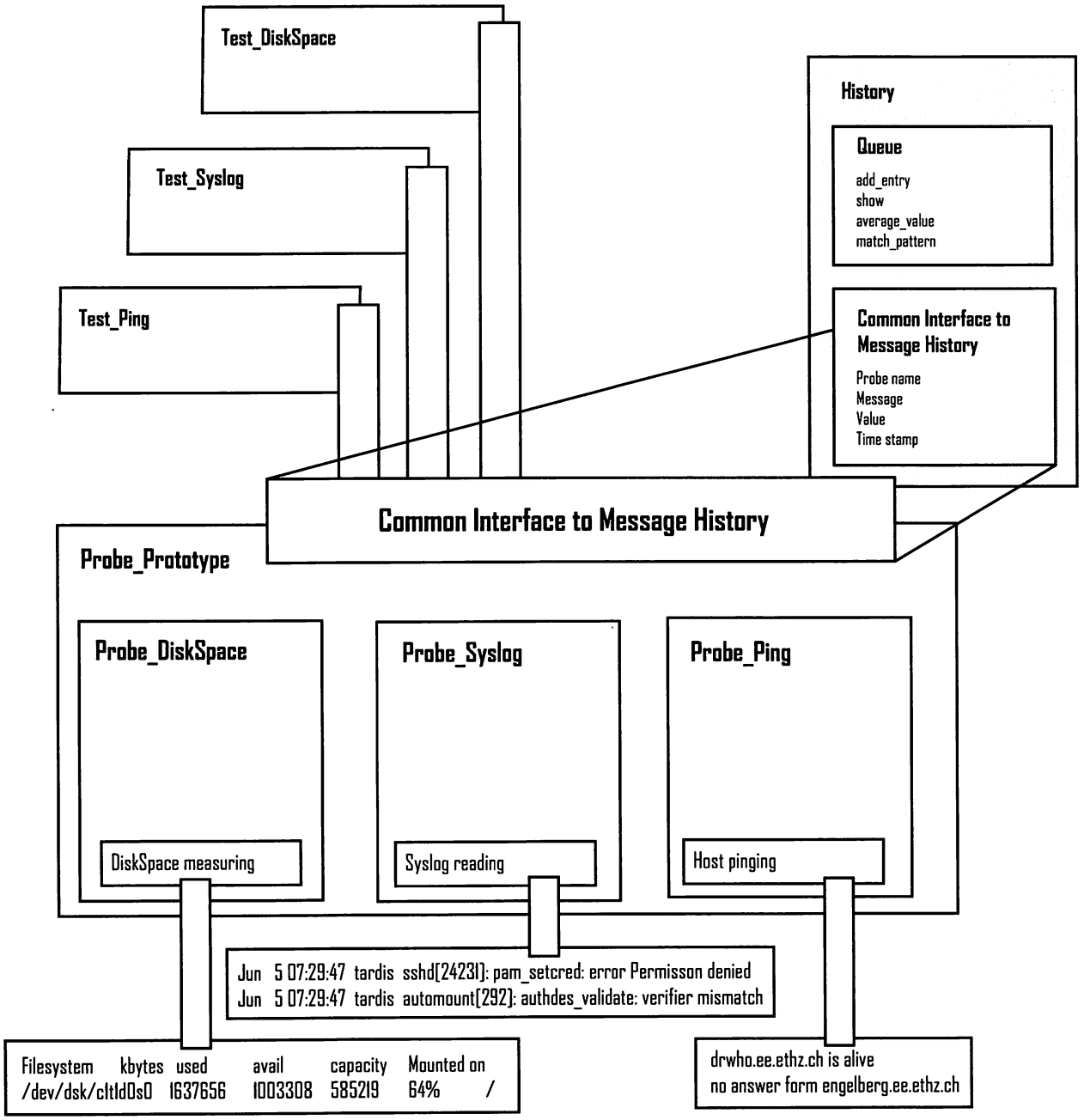


Figure 2: Objects.

- **Modularity**: Not everybody has access to the vendor's source code for system utilities that produce syslog entries. This makes it really difficult to enhance or reuse these utilities of Swatch.

Spong

- **Configuration**: Spong uses one global configuration file which sets internal values. If a specific client needs different settings you have to 'override' the default configuration with a new file. A monitoring system should be managed centrally even if it is a distributed system. The concept of grouping hosts for network tests should have been applied to local tests, too.
- **Extensibility**: You can write new tests by implementing new plugins. The measured state of the test has be mapped to status colors *red*, *yellow* or *green*.
- **Modularity**: Spong does not offer support reusing of implemented tests or parts of it. The plugin-system does not define an interface for intercommunication of different tests. A separation of data measuring and data analyzing would be a step to go towards modularity.

Results

In the process of evaluating the other products we found lots of fascinating concepts and ideas. But no tool had a really flexible framework for writing new tests. The framework we envisioned would handle all the basic functionality of a monitoring system like *execution time*, *message handling* and *internal communication*. Such a toolkit helps to implement new monitoring tasks much faster as the developer can focus on the functionality of the new monitoring task. We have not found a system which separates data acquisition and data analysis, allowing the implementation of reusable monitoring tasks.

In the end no tool fulfilled our criteria to a degree which encouraged us to add the missing bits to an existing package, so we decided to implement the tool ourselves.

Gossips Design

Architecture

Gossips is a object oriented framework written in Perl. The software is designed as a distributed client/server architecture where all clients report to a central server. Gossips is configured through a central configuration file and controlled via a command-line interface. A message handling system on the server notifies the system manager about system-state-changes. This concept is similar to the messaging of cfengine [7]. Cfengine writes a message when it changes something on a system and gossips notifies the system manager if and only if a state-change occurs in the system. Thus there is no need for a graphical display of the system status, as most of the time nothing changes. For long-time monitoring of system status, a tool as for example RRDtool [8] can be used within the gossips frame work. (See Figure 9 for an example.)

Probes and Tests – Separating Measurement from Analysis

Each participating client runs a gossips monitoring process. Each gossips process consists of a set of *probe* objects to acquire data about the state of the local system or anything else you want to observe. Data from these probes is then analyzed by a set of *test* objects. Each test can subscribe to any number of probes. This separation of data collection and data analysis was an important step toward simplifying the design, implementation and reuse of new monitoring components for the gossips system (see Figure 2).

Gossips uses a scheduler similar to the one implemented in pikt. The scheduler manages the execution of all tests and probes within a gossips instance. It executes the probes periodically. When a probe finds new data it adds all tests which have subscribed to its data to the scheduler. When a test is executed, it accesses the data acquired by the probe together with a history of old data. The test evaluates the data and decides about the state of its target.

States – Describing Systems or Services

Simple Monitoring Tasks

The generation of states relies on the data gathered by the probes. States describe the condition of a system or a service. It is up to the developer of a test to decide what states best describe a certain system. Simple things like *working/broken* are possible but also more complex approaches with many different states of operation. For example, if *free disk space* is monitored, a system manager needs to know when a certain *threshold* is reached. Additionally, it would be helpful to predict if a disk will fill up in the next hour. Because the test does not only see current data from a probe but also data collected earlier, it can make much more in-depth decisions as if it had only access to the latest measurements. This feature makes is possible to do trend analysis of measured values. In the *free disk space*-test just mentioned, all the data that was accumulated is used to calculate an approximate time when the disk fills up. The *free disk space*-test can then use the following states to decide the condition of a disk, all the values in this example are thresholds:

- Everything okay with disk /scratch
- less than 200 M on disk /scratch
- less than 30 minutes until disk /scratch full
- less than 200 M on disk /scratch and less than 30 minutes left until disk full

Combining Monitoring Tasks

By assigning several probes to a test, a next level of defining states is reached. An ftp-test, for example, could just monitor an ftp connection to a host. It could use simple states like *working* or *broken*. The client monitor might also test the 'pingability' of a host. When the observed host crashes or is rebooted gossips

would then come up with two messages, one noticing the broken ftp connection and the other that the host is not alive. This is redundant information. The important information at this time is that the host is not alive.

Therefore an ftp-test should be implemented that checks the ftp connection with an ftp-probe and simultaneously evaluates the 'pingability' of a host using a ping-probe. The test is then able to access information on status messages of these two probes and use states like:

- Everything okay with ftp connection to tardis
- no ftp connection to tardis
- no answer from tardis

A test that subscribes to several numbers of probes allows very comprehensive state assessments. As each instance of gossips is able to decide about the state of the system it monitors, it will only talk to the central server if something interesting happens (a state-change). Because normal operation is much more common than problems, this approach helps to keep communication between clients and server down to a minimal level.

Configuration

Central Configuration

One of the main design goals of the project was to keep the configuration files in one central location. Therefor gossips uses a central file for test parameters. Systems like Big Brother or Spong with their local config files for each client are much more cumbersome to change. If the parameters of a test must be edited for each client the system manager has to do lots of editing. With the complexity reducing group-design of gossips the system manager only has to edit some lines in the test.cfg file.

Distribution of the Configuration

All instances of gossips get their configuration from a central configuration. When a gossips process on a client is started, it contacts the server and asks it for its configuration. The server can also push new configurations out to the clients as each client connects to the server in a regular interval to assure the server that it is still alive.

host.cfg

Every host in the IT-environment is subscribed to groups. These groups describe hardware, network and organizational setup of a host. This design is similar to the class concept of cfengine. The difference between the two designs is that gossips uses a separate file to define a host-group relation whereas cfengine lets the host derive its memberships to the defined groups. This was made to be flexible enough to define abstract terms like *department names*, *computer room names*, *institutes* or even *disk size* as groups. See Figure 3 for an example of a host configuration file.

test.cfg

Tests are configured by assigning parameters to groups. This allows to define a network wide configuration and also the specification of test parameters for a particular host. This is a similar approach as the configuration model of cfengine. For example every host is subscribed to a group called 'ee', meaning it is located in the department of electrical engineering. All of these hosts receive the same test parameters when the parameters are assigned to the 'ee' group. In Figure 4 part of a test configuration file is shown. Each test configuration section starts with its name encapsulated by three asterisks (***). Lines starting with '+' build a subsection to attach parameters to groups.

```
*** HOSTS ***
server     server,ignore
tardis     ee,tardis,sun,2cpu,link,ignore
engelberg  ee,isg,sun,1cpu,ignore
nova       ee,isg,sun,2cpu,ignore
jabba      ee,jabba,sun,1cpu,ignore
tardis-a4  ee,tardis-a,sun,1cpu,4gb,ignore
```

Figure 3: host.cfg – file.

```
*** Test_DiskS ***

+ ignore
  run = no

+ sun
  disk1 = scratch::100M::20min
  disk2 = tmp::100M::20min

+ sun&4gb
  disk = default::50M::30min

*** Test_Load ***

+ ignore
  run = no

+ 2cpu
  period = 60sec
  timeavg = 30min
  proclim = 3proc

+ 1cpu
  run = no
```

Figure 4: test.cfg – file.

Gossips can also handle more complex group structures in the test configuration. By chaining several groups with '&' it is possible to assign very specific parameters to selected hosts. If parameters for a group 'sun&4gb' are defined gossips would apply this configuration only for hosts belonging to both groups 'sun' and '4gb'.

Merging the Configuration Information

The server process reads the configuration files and build an internal structure by merging the information. The merging algorithm searches in each test section of the config file shown in Figure 4 for a matching constellation with a host by seeking from the bottom to the top. On the top of every test section is a group called 'ignore'. It has the parameter 'run = no' which deactivates the test for a group. As you can see in Figure 3 all hosts are member of the 'ignore' group. If the merging algorithm finds for a host no other match than the 'ignore' group, the test is deactivated

for the host. If no match can be found at all, meaning, a host is not a member in the 'ignore' group, gossips will tell the system manager to review his configuration files.

Based on the information available in the configuration file fragments shown in Figures 3 and 4 the host 'tardis-a4' would receive the configuration shown in Figure 5.

```
tardis-a4:
     Test_DiskS: sun&4gb
             default::50M::30min
```

Figure 5: Configuration of tardis-a4.

The Knowledge Base

One of the functions of the gossips server is to provide a message handling system which notifies the system manager of state-changes found by tests running on the clients.

Because gossips reacts to state-changes and not to system conditions it will only report a broken disk once. If the disk breaks, this is a state-change, and gossips will report it. The disk will only be reported again when the state of the disk changes (e.g., miracle healing).

Depending on the nature of the state-change, the solution to the problem might not be obvious. When a problem occurs for the first time, there is no helping it, someone has to get to the bottom of the problem and find a solution. Once the solution is found, gossips allows to attach a description of the solution to the original message. Gossips stores this information in its knowledge base. When this particular state-change occurs again, gossips will not only inform the system manager about the new state, but will also tell about the solution which was found last time.

It is possible that in some cases many different causes will result in the same state-change. Lets look at a hard drive which is running out of space. When this happens for the first time, the system manager will add a description of the problem to the knowledge base. If the state-change occurs again at a later stage and the system manager finds a different cause for the problem, the knowledge base entry can easily be edited to explain the second possible cause as well. Otherwise the trigger can be adjusted to match the state-change more closely.

If the system manager notices that a certain problem occurs again and again, gossips could be used together with cfengine, which is able to do reparations or rebuild configurations.

Message Handling System

Gossips does not maintain a fancy web page with red and green icons indicating the system health. Normally it is quiet and leaves the system manager alone. Only if a problem occurs gossips searches its knowledge base and initiates a message to the system manager about the new state of the system or service. The communication module at the moment uses email, but it can easily be extended to talk over other transports, e.g., a pager. Visual monitoring tasks can be implemented for long-time monitoring by using RRDTool as graphic library (see Figure 9).

Implementation of Gossips

Startup of a Process

Let's start at the beginning and see what happens when you start the monitoring system. A gossips

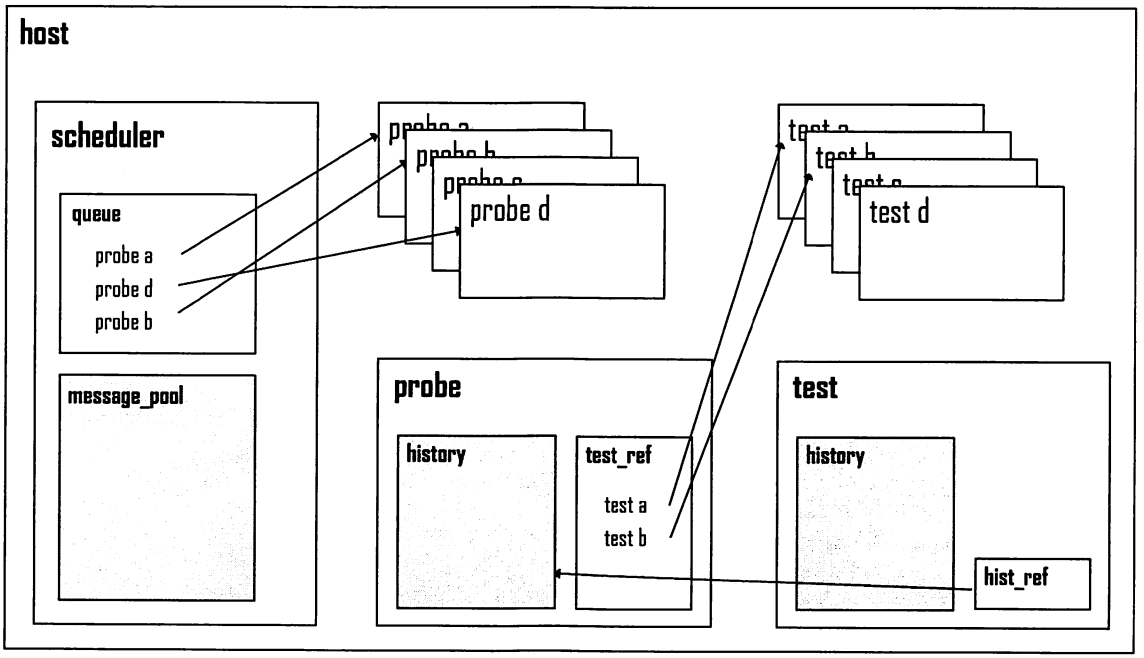


Figure 6: Structure of a host.

distribution contains two shell scripts which are designed to be executed as *init.d scripts*. The startup scripts *gossips-client-control* and *gossips-server-control* will each start the related process as daemons. Both scripts handle the command-line arguments *start*, *stop* and *restart*.

Internal Organization of a Gossips Process

Server and Client Modules

There is a single main gossips program. By using different startup parameters it loads either the server or the client modules. Every gossips process has the same objects, a *scheduler*, as well as several *probe*- and *test*-objects. In Figure 6 the internal structure of a host is illustrated. The next subsection will describe the function of each object.

Objects in a Gossips Instance

The scheduler object manages the internal operation of a gossips process. It uses a queue to control the *firetime* of probes and tests. Every probe object consists of a period. When a probe has finished its execution the scheduler puts it back into the queue and it will be re-executed after the specified time interval.

Every object in a gossips instance has a history object attached. The history object of the scheduler is called *message_pool*. To save the states of the related object the history uses a stack of constant length. In addition, the history supplies methods to evaluate its contents. For example, it provides a trend analyzing method which calculates a *gradient* of the numerical values stored in the history.

The probe objects gather the data for the monitor system. The data is stored in the attached history and accessible for the test objects through a reference. The test objects which evaluate the measured data are referenced in the probe object. At the end of its execution the probe inserts all the test objects that are subscribed to it into the scheduler queue. If a test is already scheduled it will not be added to the queue again.

Client/Server Communication

The gossips client/server architecture is implemented with probes and tests. The client and the server both use modules to communicate with each other. Each module uses a probe and a test object to implement its functionality. In Figure 7 the client/server architecture and the relation to the system manager is shown.

When a test on a client detects a new state, it pushes the related message into the message pool of

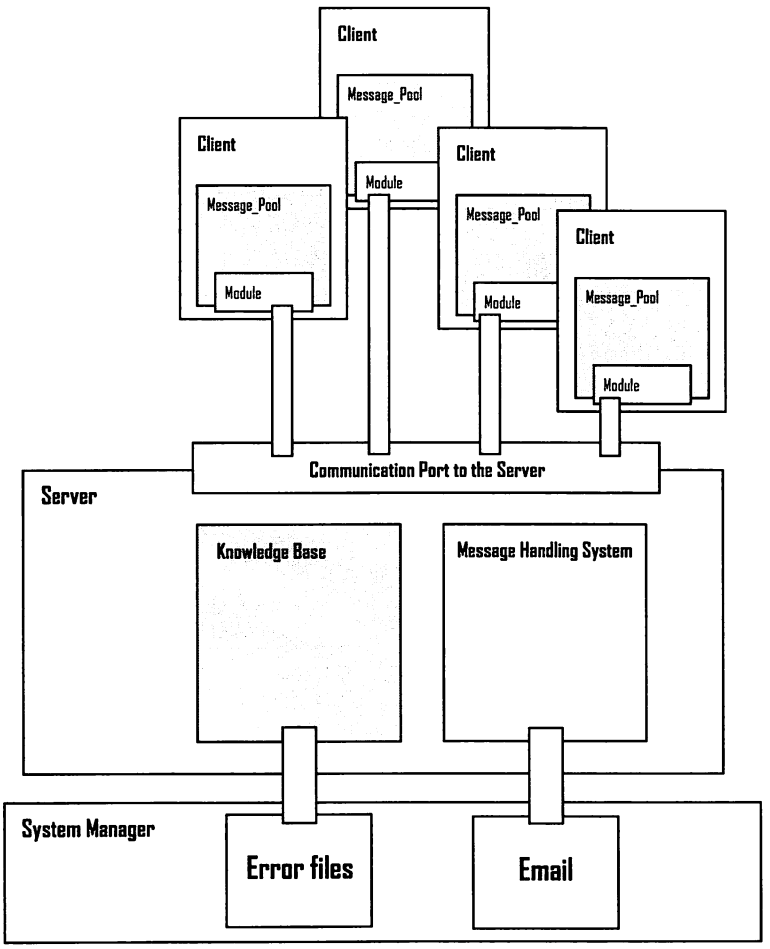


Figure 7: Client/Server architecture.

the scheduler. A probe monitors the message pool. If a new message is put in the message pool the probe schedules a test that connects to the server and forwards all new messages from the message pool to the server.

On the server a probe listens for client connections. The client authenticates itself using a *challenge/respond*-module. The communication socket itself is not encrypted by default, but it is possible to modify the client/server-modules to use the *IO-Socket-SSL*-perl-module which provides *SSL* functionality.

Current Gossips Distribution

The current gossips distribution is not just a monitoring toolkit. The current release of the package consists of an installation system, the gossips base classes, several monitoring tasks, and full documentation. Figure 8 shows a listing of the currently implemented monitoring tasks.

Extending

Base Classes

One of the main reasons for designing gossips as an object oriented framework was to define a clear and simple interface for adding new tests and probes. Gossips comes with base classes for tests and probes including several methods. The base classes provide all the communication infrastructure required for tests and probes. They also handle the scheduler as well as a few other essential gossips services.

The first step to build a new monitoring task is to separate data collection from data evaluation. Data collection is done with the probe object that measures a device or a service. The evaluation of the collected data is done by the test object. Both objects are instances of a basic test and a probe class.

Adding Probes

Probes often use UNIX-commands to collect data. Gossips supports the execution of external commands through a method called 'safe_run' which kills any started process if it does not complete within a given amount of time.

The main method of a probe object is the 'my_script'-method. It must be overridden when inheriting from the basic probe class. The job of the scheduler is to execute the 'my_script'-method. (See Figure 10 for an example of a method that pings hosts.)

Adding Tests

It is a bit more complex to implement a new test class. Again the main method that is called by the scheduler is named 'my_script.' Additionally, a method must be added that defines a language to parse the desired parameters from the configuration file and one that links these parameters with the probes and the test. Those two methods are explained in the next section.

The new test object will determine a certain state from the data acquired by the probe. This state is the return value of the main method 'my_script' (see Figure 11 for an example of a method evaluating ping measurements). In this example the 'my_script' method uses a pattern analyzing feature of the history object. This

Measurement Classes	
Probe_Logfile.pm	gathers lines of a logfile in its history.
Probe_DiskS.pm	measures free disk space using the UNIX command 'df -l'.
Probe_Load.pm	measures the load on the local host using the UNIX command 'uptime'.
Probe_Ping.pm	pinging hosts. (using 'ping')
Probe_MultiPing.pm	pinging hosts more than once. (using 'fping')
Probe_FTP.pm	checks ftp-connections to a host.
Probe_FileSize.pm	measures the size of a given file and returns the filename and its size in kilo bytes.
Analysis Classes (basic)	
Test_Logfile.pm	analyses logfiles using regular expressions.
Test_DiskS.pm	checks if there is enough space and enough time before a disk is filled up. It uses threshold values for available space on a disk and a time window in which the disk should not fill up.
Test_Load.pm	checks if the load of a local host is critical over a given period of time.
Test_FTP.pm	checks the ftp-connections and the pingability of a host simultaneously.
Test_MailWatcher.pm	checks if the size of the INBOX-file is beyond a given threshold. It sends an email to the respective user if the mailbox is too large.
Analysis Classes (graphical/using RRDTool)	
Test_DiskGraph.pm	builds a html-page with graphics which display free and used disk space of local disks.
Test_LinkUp.pm	draws a graph of the round trip times between the localhost and a given remote host (see Figure 9).
Test_MailGraph.pm	draws graphs about sent, received, bounced and rejected mails of your mailserver.

Figure 8: Features of a gossips distribution.

method only returns the first message of the history if it was repeated at least twice in a row. This feature forces the test to verify a received probe message. The state is only returned when it was confirmed once again. This test directly uses the returned messages of the ping command as states. The ping command of a Solaris distribution returns messages like *hostname is alive*, *no answer from hostname* or *ping: unknown host hostname*. On a Linux system the 'my_script' method would be implemented differently.

```
sub my_script {
  my $self = shift;
  my $history = shift;
  my $message = $history->show_message();
  return $history->first_entries_eq(1);
}
```

Figure 11: Test_Ping.pm: my_script-method.

The history object provides several methods to handle the collected data of the probe. It has methods to show the content of history entries. A history entry contains the name of the owner object, a message field, a value field and a time-stamp. Value fields could, for example, store available disk space in a test monitoring a hard disk.

The history also provides methods that evaluate its value fields. One example is an *average*-method that calculates the arithmetic mean of all values in the history entries. The history provides the *gradient*-method to be able to predict trends of measured values. This method calculates a gradient using the values from the history entries along with its time-stamp.

The result of the 'my_script'-method is the identified state of the measured service. Gossips then decides if the result is a state-change. If it is, gossips puts the state message into the message pool of the scheduler object.

Defining the Configuration

The configuration system of gossips gives the test developer the freedom to define his own 'parameter style.' Two methods are required in the test module to define the syntax of the parameter and the assignment of parameters to the test and the probes.

A 'my_syntax'-method defines the syntax of the test parameters in the configuration file seen in Figure 4. Figure 12 shows the corresponding 'my_syntax'-method of the 'Test_Load'-class.

- [1] defines the parameter key 'run'.
- [2] assigns a syntax to 'run'. The syntax is given by a regular expression (/^no$/). For the key 'run' the parser just accepts the line 'run = no'. Otherwise it throws the error message 'wrong run value'.
- [3] defines the parameter key 'period'.
- [4] assigns a syntax to 'period'. The regular expression (/^\d+sec$/) is the syntax. With this configuration the parser accepts only lines starting with a number and ending with the

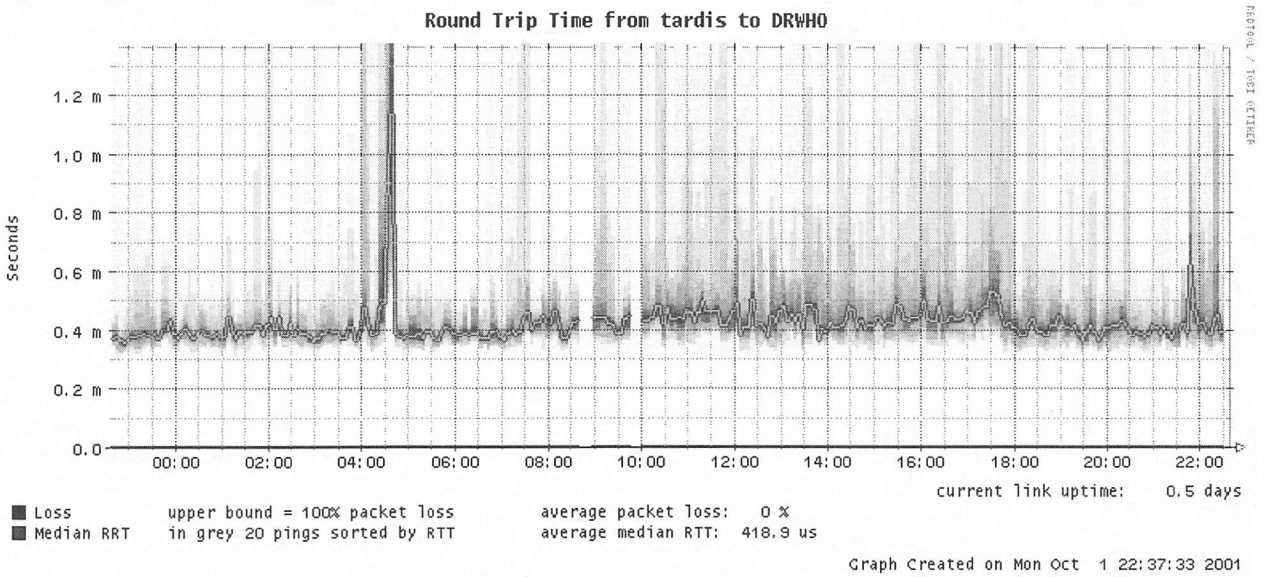


Figure 9: Example of a RRDTool-Graph used in gossips (Test_LinkUp).

```
sub my_script {
  my $self = shift;
  my $target = $self->argument;
  my $message = $self->safe_run("/usr/sbin/ping $target 5");
  return $message;
}
```

Figure 10: my_script-method of Probe_Ping.pm.

identifier 'sec'. On failure it will respond "syntax error in 'period' parameter".

To assign the different parameters to the test and probes the developer has to implement a 'my_struct'-method. Again, the test base class offers methods to define these relations.

In the 'my_struct'-method of the 'Test_ Load'-class seen in Figure 13 the developer first adds the keys defined in the configuration file.

- [A] defines the key 'period'.
- [B] adds a 'filter' to the first parameter of key 'period'. The filter '/^(\d+)sec/' is used to extract the information from the parameter. In this case the filter cuts off the identifier sec'.
- [C] defines the key 'timeavg'.
- [D] adds the filter '/(\d+)min/' to the first element of the key 'timeavg'. This filter lets pass just the minutes which are defined.
- [E] adds the probe 'Probe_Load' to the test.
- [F] links the first argument of key 'period' to the period of the probe 'Probe_Load'. This command sets the period of a probe.
- [G] links the first argument of key 'timeavg' to the test.
- [H] links the first argument of key 'proclim' to the test.

Figure 14 illustrates the relations between the configuration parameter, the parser, the test and the probe object.

Defining the States

The generation of states relies on the data gathered by the probes. For a simple test like a ping test the collected data already defines reasonable states like '*host is alive*' or '*no answer from host.*' In more difficult cases the test developer has to define his own set of states in the test class.

The main job of the 'my_script'-method in the test module is to handle the messages in the history of the probe. The message should be mapped to logical states. The definition of a sensible set of states is essential for successful monitoring. The type of information that flows from the history into the states is restricted in some points. Remember that gossips supplies state-changes. States should express if they are *good* or *bad*. By using such states gossips is able to tell the system manager if a monitored service just changed to a *bad* state. If gossips monitors, for example, a hard disk by collecting the free disk space it should use a threshold value. With such a value it can define states like '*Everything okay with disk /scratch*' when the free disk space is larger then the threshold or '*less than 100 M on disk /scratch*' if the free disk space shrinks under the defined mark of 100 MB. The important point for the design of states is that they should not contain changing elements like *actual disk-size*, *uptimes*, etc. Otherwise gossips will generate lots of state-change messages overwhelming the system manger. Gossips provides the possibility to log

```
sub my_syntax {
    my $self = shift;
[1] $self->add_syntax_key('run');
[2] $self->add_syntax_to_key('run','/^no$/',"wrong 'run' value");
[3] $self->add_syntax_key('period');
[4] $self->add_syntax_to_key('period','/^\d+sec$/',
                             "syntax error in 'period' parameter");
    $self->add_syntax_key('timeavg');
    $self->add_syntax_to_key('timeavg','/^\d+min$/',
                             "syntax error in 'time average' parameter");
    $self->add_syntax_key('proclim');
    $self->add_syntax_to_key('proclim','/^\d+\e.?\ed*proc$/',
                             "syntax error in 'proc limit' parameter");
}
```

Figure 12: Test_Load.pm: my_syntax-method.

```
sub my_struct {
    my $self = shift;
[A] $self->add_key_to_struct('period');
[B] $self->add_filter_to_parameter('period','/^(\d+)sec/',1);
[C] $self->add_key_to_struct('timeavg');
[D] $self->add_filter_to_parameter('timeavg','/^(\d+)min/',1);
    $self->add_key_to_struct('proclim');
    $self->add_filter_to_parameter('proclim','/^(\d+)proc/',1);
[E] $self->add_probe_to_struct('Probe_Load');
[F] $self->link_key_elem_to_probe_period('period',1,'Probe_Load');
[G] $self->link_key_elem_to_test('timeavg',1);
[H] $self->link_key_elem_to_test('proclim',1);
}
```

Figure 13: Test_Load.pm: my_struct-method.

changing values like '*free disk space.*' These values can be submitted to the server along with the state message. The server then stores these values in a log-file associated with the corresponding knowledge base file.

Conclusions

The distributed architecture of gossips builds a scalable monitoring system. Through its flexible and central configuration environment, together with its command-line module, gossips is easily maintainable. The object oriented design of gossips builds a flexible and well defined framework for developing new monitoring tasks. The concept of separating data acquisition and data analysis makes defined monitoring tasks reusable and provides the possibility to build combined tests. The knowledge base allows to archive solutions to known problems in one place and to integrate the knowledge of the system manager.

By including cfengine, gossips could be extended into a automated repair tool. Development of an SNMP-probe-class would extend the monitor software to a low level device monitor.

Availability

Gossips source and documentation along with its monitoring tests are available on the web-page http://isg.ee.ethz.ch/tools. There is a mailing list on gossips. Send an email with subject: *subscribe* to gossips-request@list.ee.ethz.ch to subscribe.

Acknowledgments

We would like to thank the following people for their feedback and suggestions: our co-workers Andreas Karrer, David Schweikert, Edwin Thaler, Christoph Wicki, Fritz Zaucker, as well as our shepherds Mark Burgess and Todd K. Watson.

Author Information

Victor Götsch is a third year Computer Science student at the Swiss Federal Institute of Technology, Zurich. After finishing his second year he started an internship with the IT Support Group of the Department of Electrical Engineering where he learned a lot about system management and spent most of his time developing the System and Service Monitor gossips. He will continue his studies in fall 2001 to get his degree in Computer Science.

Albert Wuersch got a degree in Electrical Engineering from the Swiss Federal Institute of Technology in 1999. He worked for nine months as a Trainee System Manager for the IT Support Group of the EE Department. During that time he designed the gossips concept and started the implementation.

Tobias Oetiker is a Senior System Manager with the above mentioned IT Support Group and has guided the gossips project over the last 18 months.

References

[1] Sellens, John, "System and Network Monitoring," *;login:*, Vol 25, No. 3, June, 2000.

[2] Case, Fedor, Schoffstall, and Davin, "A Simple Network Management Protocol (SNMP)," *RFC 1157, SNMP*, May, 1990.

[3] MacGuire, Sean, "Big Brother, a tool for proactive network monitoring," http://www.bb4.com.

[4] Hansen, S. E., E. T. Atkins, "Automated System Monitoring and Notification With Swatch,"

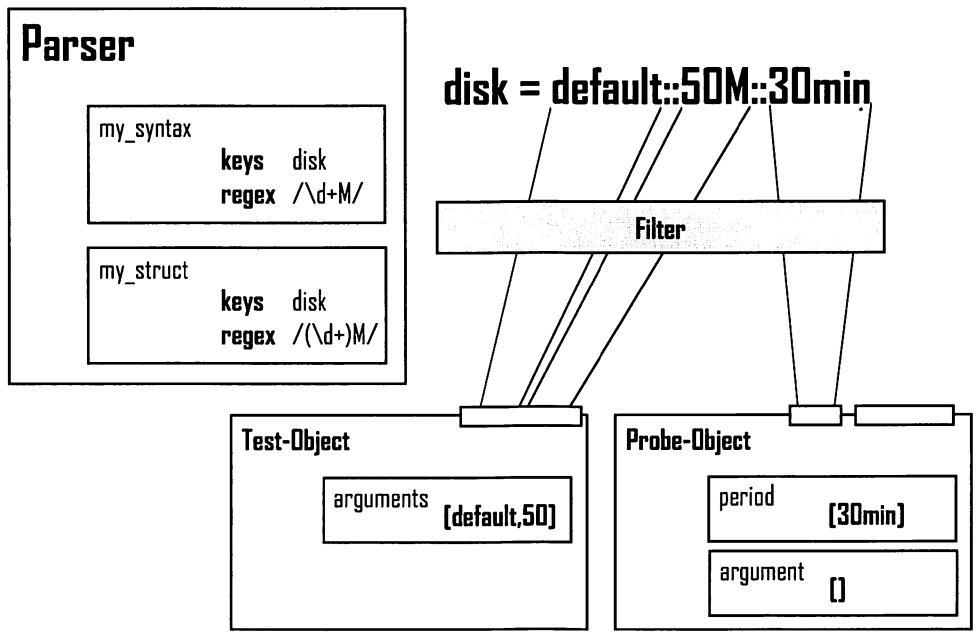


Figure 14: Filtering of test parameter and linking them to the test or probe object.

Proceedings of the Seventh Systems Administration Conference (LISA VII), p. 145, USENIX Association, Berkeley, CA.

[5] Johnson, Stephen L., "Spong – Systems and Network Montoring," http://spong.sourceforge.net.

[6] Osterlund, R., "PIKT: Problem Informant/Killer tool, *Proceedings of the Fourteenth Systems Administration Conference (LISA XIV)*, p. 147, USENIX Association, Berkeley, CA.

[7] Burgess, Mark, "Cfengine: A Site Configuration Engine," *USENIX Computing Systems*, http://www.iu.hio.no/cfengine, Vol 8, No. 3, 1995,

[8] Oetiker, Tobias, "RRDTool, The Round Robin Database Tool for Long Time Monitoring," http://people.ee.ethz.ch/~oetiker/webtools/rrdtool.

The CoralReef Software Suite as a Tool for System and Network Administrators

David Moore, Ken Keys, Ryan Koga, Edouard Lagache, and kc claffy – CAIDA

ABSTRACT

Until now, system administrators have lacked a flexible real-time network traffic flow monitoring package. Such a package must provide a wide range of services but remain flexible enough for rapid in-house customization. Existing passive data collection tools are typically narrow in scope, designed for specific tasks from packet capture (tcpdump [9]) to accounting (NeTraMet [4]). In response, CAIDA has created the CoralReef suite designed to provide network administrators and researchers with a consistent interface for a wide range of network analysis applications, from raw capture to flows analysis to real-time report generation. CoralReef provides a convenient set of passive data tools for a diverse audience.

CoralReef is a package of device drivers, libraries, classes, and applications. We briefly outline the architecture and provide relevant case studies and examples of CoralReef's use as applied to real-world networking situations. We will show how CoralReef is a powerful, extensible, and convenient package for network monitoring and reporting.

Introduction

With the growth in traffic volume and increasing diversity of applications on the Internet, understanding and managing networks has become increasingly difficult and important. To this end we have created the CoralReef passive traffic monitoring suite, which allows network users, administrators, and researchers to measure and analyze network traffic. The CoralReef software suite is a comprehensive collection of tools developed by CAIDA to collect, store, and analyze traffic data. CoralReef can be deployed on a dedicated monitor host using data capture cards that tap a fiber optic link, or on virtually any UNIX system without special hardware using libpcap interfaces. CoralReef software handles everything from the low level details of cell and packet capture to the production of high level HTML reports in near real-time. Network and system administrators can use the CoralReef suite to monitor and interpret a wide range of observed network behavior.

CoralReef evolved from OCXmon monitors, developed jointly by MCI and NLANR [1, 2]. The OCXmon monitors ran on MS-DOS, could only monitor ATM links, and provided only basic cell capture (in device-dependent format) and limited flow summary capability. CoralReef runs on UNIX, and supports device independent access to network data from OCXmon hardware, native OS network interfaces, and trace files; programming APIs; a variety of bundled analysis applications; and greater flexibility in remote access and administration. CoralReef is developed and tested under FreeBSD, Linux, and Solaris, although specialized hardware drivers are not available for all operating systems. CoralReef has two releases, a "public" non-commercial use version and a version available only to CAIDA members. Both versions implement the same set of libraries and APIs, but the members-only version incorporates performance and operational enhancements geared toward CAIDA members. What makes CoralReef unique is that it supports a large number of features at many layers, and provides APIs and hooks at every layer, making it easier for anyone to apply it in unanticipated ways and develop new applications with minimum duplicated effort.

Because commercial software tools lack sufficient flexibility, network administrators often develop their own network analysis tools, typically based on tcpdump [9]. A part of tcpdump is the library libpcap [13] which provides a standard way to access IP data and BPF (Berkeley Packet Filter) devices. The tcpdump tool also has a packet data file format (pcap) which has become a *de facto* industry standard. Several network analyzer tools are built on top of libpcap, such as the Ethereal [7] protocol analyzer and NeTraMet (RFC 2722 [5] and RFC 2724 [8]), which are geared toward long term collection for metering and billing. Other network analysis tools include the MEHARI [12] ATM/IP analysis system; Narus [16] for long-term workload and billing; the DAG ATM/POS capture cards and software [19] by the WAND group at the University of Waikato, New Zealand; Clevertool's netboy [6]; Network Associates Sniffer Pro [3]; and NIKSUN's NetVCR [17].

An Overview of the CoralReef Software Suite

CoralReef is a package of libraries, device drivers, classes, and applications written in, and for use with, several programming languages. The overall architecture and programming interfaces are described in a separate paper [11] and are not covered here. Figure 1 shows an overview of the relationships between

CoralReef applications. Detailed descriptions of the software tools can be found at the CoralReef web site (http://www.caida.org/tools/measurement/coralreef/).

Most CoralReef applications fall into one of two categories: those with names beginning with "crl_", which operate on raw packet data; and those with names beginning with "t2_", which operate on aggregated flow data. We will refer to these groups of applications as crl_* and t2_*, respectively. Sources of raw data include custom Coral drivers for special collection cards, the libpcap library for commodity network interfaces, and trace files generated by crl_trace, tcpdump, or other software.

Raw Traffic Applications

All of the crl_* applications take a common set of command line and configuration options. These options include stopping after a specified number of packets or ATM cells or after a specified time duration; link specific parameters; filtering by ATM virtual channels; number of bytes to capture from each packet; and debugging level. Additionally, applications that operate on packets can filter their input with BPF (tcpdump) filter expressions. Applications that operate at regular time intervals have a common syntax for specifying the interval size.

Pure utilities:

- crl_trace: captures network traffic to a .crl trace file
- crl_info: reports hardware and link configuration details of a trace file
- crl_time: outputs timestamps and inter-arrival time information for packets or ATM cells
- crl_encode: encodes the IP addresses in a .crl file to protect privacy
- crl_to_*: captures network traffic or converts trace files to other file formats

Simple tools:

- crl_print: prints headers and payloads of ATM cells
- crl_print_pkt: prints multiple layers of protocol headers and payloads of packets
- crl_rate_layer2: at regular time intervals, outputs cell count and bit rate for each ATM channel
- crl_rate: at regular time intervals, outputs IPv4 and IPv6 packet and byte counts, and counts of non-IP packets

Static Reports:

- crl_hist: reports packet and byte counts by IP length and protocol, port summary matrices for TCP and UDP, fragment counts by protocol, packet length histograms for the entire trace and for a list of applications, and the top 10 source and destination port numbers seen for TCP and UDP traffic
- crl_bycountry: reports the amount of traffic flowing to and from networks, and between networks, ASes, and countries

Specialized Utilities:

- crl_portmap: captures all packets from any hosts that connect to another host's portmap port
- crl_flow: at regular time intervals, aggregates packet data into flows by source and destination IP addresses, protocol, and source and destination ports

Traffic Flow Applications

The t2_* applications operate on tables generated by crl_flow or other t2_* applications, with the same time intervals.

- t2_report: generates HTML summary reports
- t2_ASmatrix: with a routing table, source and destination ports
- t2_top: sorts a table by packets, bytes, or flows, and displays the top N entries

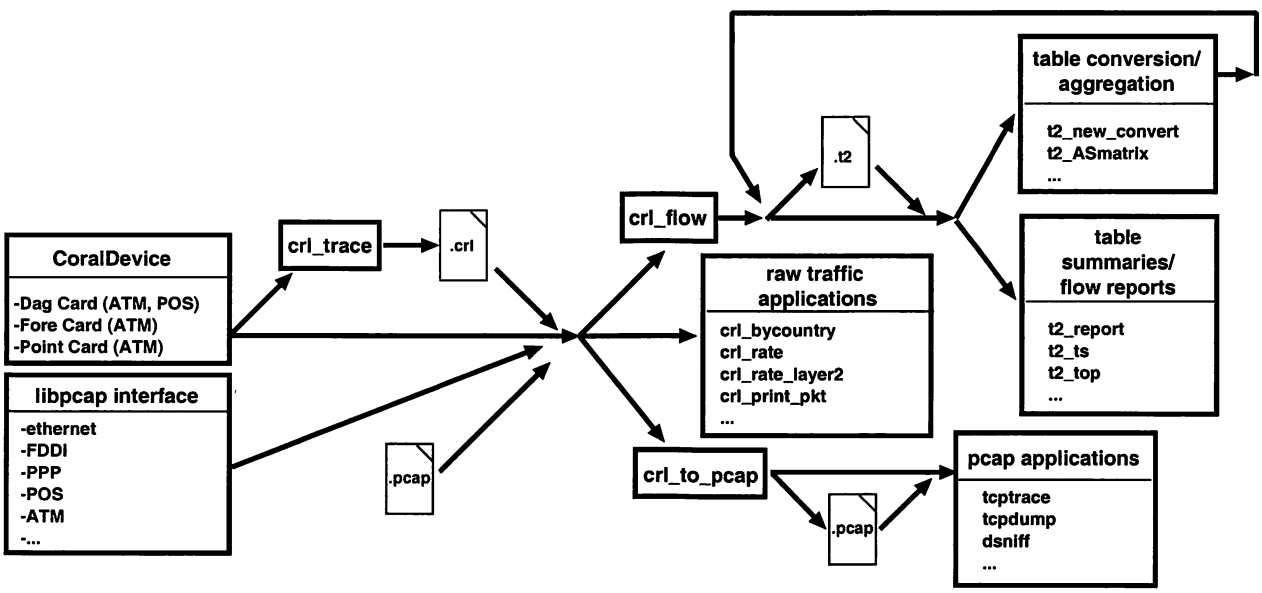


Figure 1: Overview of CoralReef applications.

- t2_ts: outputs counts of IP packets, bytes, and flows
- t2_convert: aggregates tables by specified keys

Other Applications

- crl_to_pcap: converts Coral traces or live data to pcap format for use with existing libpcap tools
- parse_bgp_dump: converts Cisco router "sho ip bgp" output to the routing table format used by t2_ASmatrix, t2_report, and crl_bycountry
- parse_bgp_mrtd: converts MRTd [15] output to the routing table format used by t2_ASmatrix, t2_report, and crl_bycountry

Libraries

- libcoral: reads trace files and live network interfaces, and provides common functionality for all crl_* applications
- Coral.pm: perl interface to libcoral
- ASFinder: maps IP addresses to AS numbers and network prefixes
- AppPorts: maps protocols and port numbers to application names
- NetGeoClient: maps IP addresses and AS numbers to geographic locations
- Tables: manipulating and processing the tables used by the t2_* applications

Using CoralReef in an Operational Setting

CoralReef can only monitor traffic that is visible to a network interface. If the network you want to monitor is a shared medium such as non-switched Ethernet or FDDI, any interface on that network is sufficient. Monitoring a link between routers or on a switched network requires directing traffic into additional dedicated interfaces, which may be either standard interfaces read via libpcap, or special hardware accessed through Coral drivers. A link can be tapped either with a physical splitter (Figure 2a) or by configuring a span or mirror port on the appropriate switch or router (Figure 2b). Note that tapping both directions of a link with splitters requires a dedicated interface for each direction.

The hardware needed depends on the utilization of the links being monitored and the amount of aggregation desired. For straightforward packet traces, the main constraint is usually disk performance and capacity; we recommend ultra-wide SCSI rather than an IDE drive. For flow collection and analysis, memory and CPU speed are more important. Individual applications in a CoralReef pipeline can run on separate machines to distribute the load. A common example of this is to run crl_flow on the monitor machine and t2_report on a different machine.

Examples

In this section, we briefly present examples of using CoralReef in an operational setting. A more complete outline of uses can be found in the CoralReef documentation or the user community mailing list.

Several -C options are common to all crl_* applications. In the following examples, we use -Ci=*time* to specify the repeated interval at which the application processes data and outputs results, and -C'filter *expression*' to specify a BPF filter expression that selects the packets to be measured. To stop the applications after a specific duration, you would use the -Cd=*time* option. The crl_* applications can read traffic from a variety of sources; in these examples, the data source is "if:fxp0", a native Ethernet interface (fxp0) read via libpcap.

IP addresses in the sample outputs have been encoded for privacy. Some output has been edited to better fit the page and for illustrative value.

Timestamps in application output are printed in UNIX timestamp format.

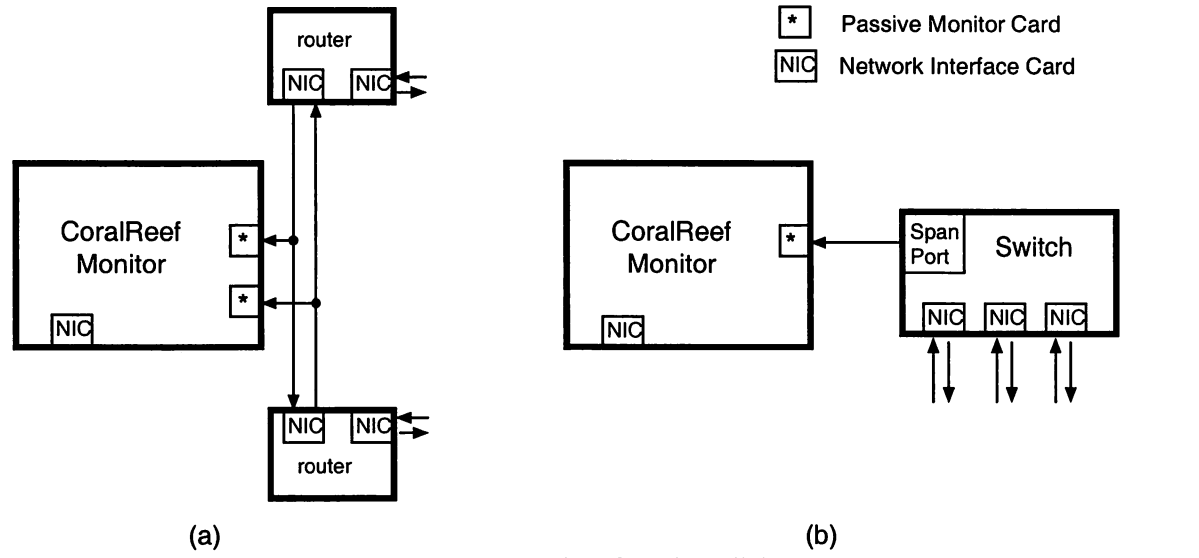


Figure 2: Examples of tapping a link.

Using crl_rate to Check Utilization of Subinterfaces

The crl_rate application counts packets and bytes on interfaces and subinterfaces at regular intervals. On Ethernet interfaces, subinterfaces are IEEE 802.1Q VLANs. On ATM interfaces, virtual channels are reported as subinterfaces. Other types of interfaces do not have subinterfaces.

Goal: Continuously Measure Traffic on a Link, in Packets and Bytes

Command line: crl_rate -Ci=10 if:fxp0
See Listing 1 for the sample output.
Explanation:

This output shows that there was traffic on seven VLANs on Ethernet interface fxp0 in a 10 second period. A similar table would be printed every 10 seconds. The "non_ip" column counts packets of protocols like ARP, AppleTalk, and IPX. The total IPv4 traffic on this link in 10 seconds was about 99.5 Megabytes, or 79.6 Megabits per second. Note that the bytes counted are those in layer three and above; bytes in lower layer encapsulations like Ethernet and ATM are not counted.

This simple example is a good way to test your CoralReef monitor and software setup to verify that the output matches your expectations.

There was only one interface in this example, labeled 0 in the output, but it is possible to monitor multiple interfaces simultaneously.

Goal: Find Out How Much KaZaA Traffic is on Your Link

Command line:
crl_rate -s -Ci=300 -C'filter port 1214' if:fxp0
See Listing 2 for sample output.
Explanation:

In this example, we were not interested in subinterfaces (VLANs), so we used the -s option to omit them. Because of high variability in these kinds of measurements, larger intervals are usually more useful. In this example, we used a five minute interval. To limit the measured traffic to KaZaA [10], we used a BPF filter to match only traffic to or from KaZaA's well-known port (1214). In the first of the two intervals shown, there were about 1.33 Gigabytes of KaZaA traffic, or 35.5 Megabits per second.

Using crl_flow to Collect Flow Data

The crl_flow application summarizes data by IP flows. In this context, a flow is identified by the 5-tuple of source address (src), destination address (dst), protocol (proto), source port (sport), and destination port (dport). A flow is unidirectional, so there will be one flow for each of the two directions of a network connection, with sources and destinations swapped.

The definition of flow termination can be chosen by a command line option. The -I option specifies that flows terminate at the end of each interval, which is the most useful definition for this kind of continuous monitoring. Other definitions are typically more useful in offline analysis of historic data, as is often needed in research situations.

A table of these 5-tuples, along with counts of packets, bytes, and flows for each, is called a Tuple_Table. The "ok" column contains a 1 if sport and dport are meaningful for the protocol and were not truncated by capturing too few header bytes. Ports are meaningful for TCP, UDP, and ICMP (for ICMP, the sport and dport columns actually contain ICMP type and code, respectively).

Goal: Continuously Collect Data on Link Use, Summarized by Hosts, Protocols, and Ports.

Explanation:

The -h option tells crl_flow to print in human-readable format. With no formatting option, crl_flow prints a tab-separated format more suitable for input to other scripts. Additionally, crl_flow -b outputs a binary

```
# time 1001975450.054545 (10.000000), packets lost: 0
# if[subif]    ipv4pkts      ipv4bytes   ipv6pkts      ipv6bytes     non_ip
0[135]                1             40          0              0          0
0[110]             2269        2536140          0              0          0
0[169]             9397        3761410          0              0          0
0[170]            40097       20640233          0              0          0
0[130]             5659        1921566          0              0          0
0[131]           118429       70553909          0              0          0
0[108]             1774          92307          0              0          0
0   TOTAL        177626       99505605          0              0          0
...
```

Listing 1: crl_rate output: traffic on a link.

```
# time 1001977053.013038 (300.000000), packets lost: 0
# if[subif]    ipv4pkts      ipv4bytes   ipv6pkts      ipv6bytes     non_ip
0   TOTAL        1999660     1330861215          0              0          0

# time 1001977353.013038 (300.000000), packets lost: 0
# if[subif]    ipv4pkts      ipv4bytes   ipv6pkts      ipv6bytes     non_ip
0   TOTAL        1956821     1215396030          0              0          0
...
```

Listing 2: crl_rate output: KaZaA traffic.

format that is readable by the t2_* applications, more efficiently than either of the text formats.
Command line: crl_flow -I -h -Ci=10 if:fxp0
See Listing 3 for sample output.

The output shown here has been edited to fit the page. Real output would have a Tuple_Table for each interface and subinterface, repeated every 10 seconds; an interface or subinterface summary preceding each table; and two additional columns in each table showing the first and last packet timestamp observed within each flow. Remember that IP addresses have been encoded for privacy.

The sample output shows one UDP DNS flow (protocol 17, port 53), one HTTP flow from a web server to a client, several HTTP flows from clients to web servers, and one large flow on TCP port 1214 (KaZaA).

Using t2_* to Monitor Utilization and Flows

Although crl_flow does some aggregation, its output is still typically too voluminous to be directly useful. The t2_* applications further aggregate or filter the output of crl_flow for more specific needs. In particular, t2_ts outputs a single line per interval summarizing the packets, bytes, and flows observed. t2_top sorts table entries by packets, bytes, or flows, and prints only the top *N*.

Most t2_* applications accept different table types as input, so the user must specify the type on the command line. crl_flow outputs a Tuple_Table; we will introduce other table types in later examples.

Goal: Continuously Measure Traffic on a Link, in Packets, Bytes, and Flows

Command line:
crl_flow -I -b -Ci=10 if:fxp0 | t2_ts Tuple_Table
See Listing 4 for sample output.
Explanation:

In this example, t2_ts prints a line for every 10 second interval, the beginning of which is indicated in the first column (time). The next three columns show the total number of packets, bytes, and flows observed during the interval. The "entries" column shows the number of table entries, which, in the case of a Tuple_Table, is equal to the number of flows. The last three columns show the average number of packets, bytes, and flows per second during the interval.

The -b option to crl_flow tells it to output in efficient binary format readable by t2_* applications. The use of this option can drastically improve performance, and is recommended when the intermediate output does not need to be read by a human.

Goal: Continuously Find Flows Consuming the Most Bandwidth on a Link

Command line:
crl_flow -I -b -Ci=10 if:fxp0 |
t2_top -b -n5 Tuple_Table

```
# crl_flow output version: 1.0 (pretty format)
# begin trace interval: 1001981488.441461
# trace interval duration: 10.000000 s
# Layer 2 PDUs dropped: 0
# IP: 101.8403 Mbit/s
# Non-IP: 0.0000 pkts/s
# Table IDs: 0[131], 0[108], 0[130], 0[110], 0[170], 0[169]
...
# begin Tuple Table ID: 0[131]
# expired flows
#src            dst          proto ok sport dport     pkts     bytes  flows
0.1.0.8         1.82.0.1        17  1    53    53        2       497      1
0.1.0.14        0.44.0.1         6  1    80  2223        4       646      1
0.3.0.148       1.95.0.1         6  1  1214 62772      125    187008      1
0.1.1.93        0.71.0.6         6  1 49200    80        3       565      1
0.1.1.93        0.71.0.6         6  1 49199    80        5       647      1
0.1.1.93        0.71.0.6         6  1 49198    80        5       647      1
0.1.1.93        0.71.0.6         6  1 49196    80        6       708      1
0.1.2.59        11.88.0.1        6  1 51643    80        6       817      1
...
# end of text table
...
# end trace interval
...
```

Listing 3: crl_flow output: Continuous data collection.

```
time                 pkts    bytes    flows  entries   pkts/s      bytes/s    flows/s
1001982746.151594  143315  97458948   6624    6624   14331.500  9745894.800  662.400
1001982756.151594  143985  98792491   6465    6465   14398.500  9879249.100  646.500
...
```

Listing 4: ts_ts output: Utilization and flows.

See Listing 5 for sample output.
Explanation:

The "KEYS" columns are the same as the keys in the input table, which in this example is a Tuple_Table from crl_flow. The -p, -b, or -f option tells t2_top to sort by packets, bytes, or flows, and the -n option specifies how many entries to print.

Using t2_* to Find Hosts Generating the Most Traffic

Often we want to aggregate the flows of a Tuple_Table by a subset of its keys. For example, we may want to count the bytes sent between pairs of hosts, regardless of their protocols and ports; or, all the packets sent from a particular TCP port, no matter what host sent or received them.

Table Type	Keys
Tuple_Table	source IP, destination IP, IP protocol, ports ok, source port, destination port
IP_Table	IP
IP_Matrix	source IP, destination IP
Proto_Ports_Table	IP protocol, ports ok, source port, destination port
Port_Table	port
Port_Matrix	source port, destination port
Proto_Table	IP protocol
AS_Table	AS
AS_Matrix	source AS, destination AS
Country_Table	country
Country_Matrix	source country, destination country
App_Table	application
VPVC_Table	vp/vc pair
Prefix_Table	prefix/masklength
Prefix_Matrix	source prefix/masklength, destination prefix/masklength
Length_Table	length

Table 1: CoralReef table types.

In addition to the Tuple_Table, CoralReef has other table types defined by different sets of keys. For example, the keys of an IP_Matrix are source and destination IP addresses, and the key of an IP_Table is a single IP address. Table 1 shows all tables and their keys.

The t2_convert application converts one table type to another by aggregating entries with common keys. A conversion operator determines which subset of input table keys to use as the keys of the output table. For example, applying the src_IP_Table operator to a Tuple_Table generates an IP_Table whose keys are the source addresses of the input table. The pkts, bytes, and flows counts of each entry in the new table are the sums of the corresponding counts of the Tuple_Table entries with the same source IP address. Figure 3 shows all the operators that can be applied to the various table types.

Goal: Find the Top Five Hosts by Bytes of Traffic Generated

Command line:

```
crl_flow -I -b -Ci=10 if:fxp0 |
    t2_convert Tuple_Table src_IP_Table |
    t2_top -b -n5 IP_Table
```

See Listing 6 for sample output.
Explanation:

The output of crl_flow is a Tuple_Table, with keys src, dst, proto, ok, sport, and dport. To aggregate those flows by source IP address, we apply the src_IP_Table operator with t2_convert. Since the flows column in a Tuple_Table is always 1, the flows column in the resulting IP_Table is the number of flows with that source IP address. Sorting this IP_Table by bytes and taking the top five entries shows the hosts sending the most bytes.

Goal: Find the Top Five Web Servers by HTTP Flows

Command line:

```
crl_flow -I -b -Ci=10 -C'filter tcp src port 80' if:fxp0 |
    t2_convert Tuple_Table src_IP_Table |
    t2_top -f -n5 IP_Table
```

See Listing 7 for sample output.
Explanation:

This is similar to the previous example, except we limit the traffic to web servers by using a filter option to crl_flow and sort by flows (-f) instead of bytes. Aggregating this Tuple_Table by source IP address and then sorting the resulting IP_Table by flows shows the

```
src             dst          proto  ok     sport    dport
#KEYS                                                      pkts     bytes     flows
0.4.0.27        0.98.0.1         6     1    46978    64671    16035    22534236      1
0.4.0.30        0.19.0.2         6     1    22       64156    2965     4230700       1
0.4.0.44        0.158.0.1        6     1    22       33222    3647     3919348       1
0.4.0.4         0.17.0.1         6     1    80       58013    1831     2702668       1
0.4.0.3         0.15.0.1         6     1    45925    20       2244     2390668       1
# end of text table
#KEYS                                                      pkts     bytes     flows
0.4.0.27        0.98.0.1         6     1    46995    64683    9311     13084084      1
0.4.0.27        0.98.0.1         6     1    46978    64671    9095     12780460      1
0.4.0.30        0.19.0.2         6     1    22       64156    3097     4417620       1
0.4.0.44        0.158.0.1        6     1    22       33222    3185     3350880       1
0.4.0.21        0.73.0.2         6     1    60971    119      1362     1915352       1
# end of text table
```

Listing 5: t2_top output: High bandwidth consumers.

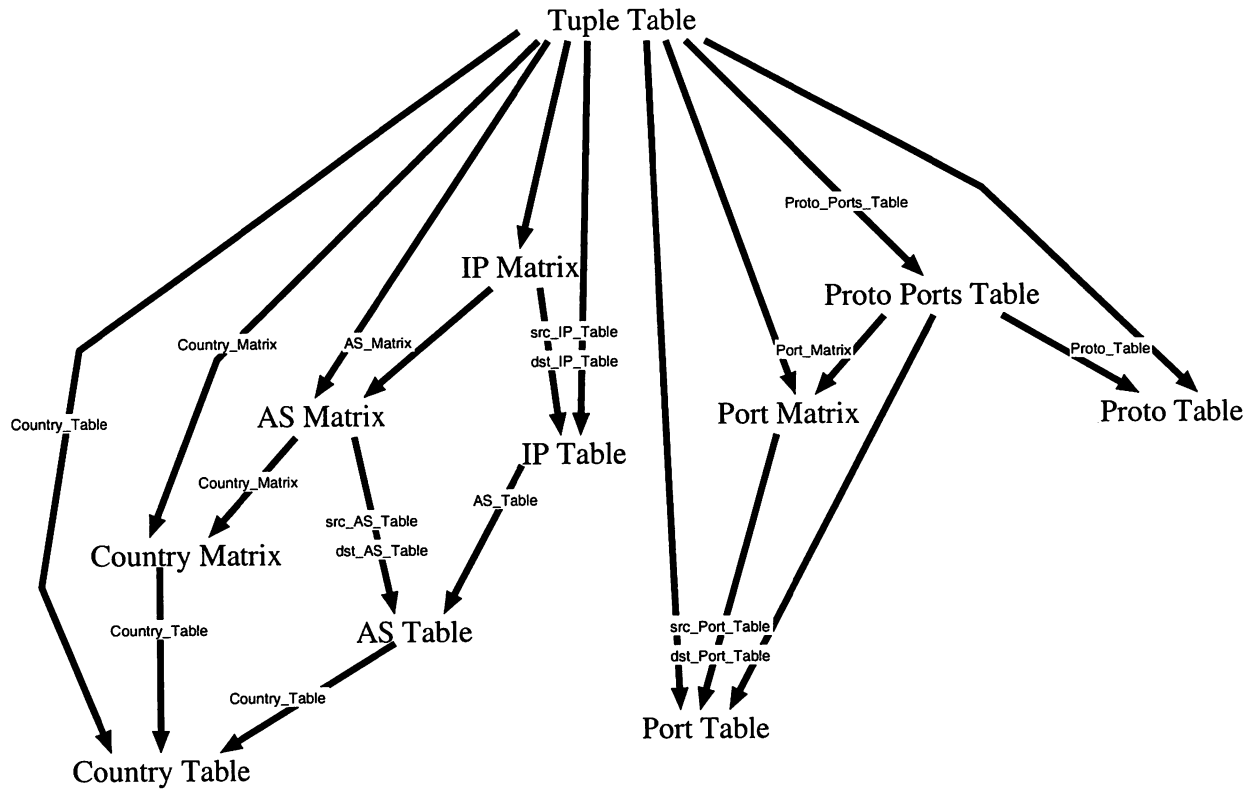


Figure 3: Table conversion operations.

```
#KEYS           pkts    bytes   flows   (top 5 sorted by bytes)
0.4.0.27        16035   22534236        1
0.4.0.21        12202   13663537        46
0.4.0.30        2965    4230700         1
0.4.0.44        3647    3919348         1
0.4.0.4         1831    2702668         1
# end of text table
#KEYS           pkts    bytes   flows   (top 5 sorted by bytes)
0.4.0.27        18409   25864829        3
0.4.0.21        13900   15515873        46
0.4.0.30        3097    4417620         1
0.4.0.44        3185    3350880         1
0.4.0.64        1347    1948443         7
# end of text table
...
```

Listing 6: t2_top output: Top five source hosts.

```
#KEYS           pkts    bytes   flows   (top 5 sorted by flows)
0.3.0.77        126     101845  23
0.1.0.108       102     52397   17
0.1.0.52        180     78166   15
0.1.0.42        8       320     8
0.1.0.182       24      1713    5
# end of text table
#KEYS           pkts    bytes   flows   (top 5 sorted by flows)
0.1.0.108       192     80733   36
0.3.0.77        131     119220  22
0.1.2.145       5       205     5
0.1.2.135       66      72221   5
0.1.0.119       17      25500   4
# end of text table
...
```

Listing 7: t2_top output: Top five web servers.

web servers with the most HTTP connections during each 10 second interval.

Using t2_* to Find Hosts Talking to the Most Hosts

Normally, the flows column in each entry of the output table of t2_convert is the sum of the flows column of the input table entries with the same output keys. But with the -F option of t2_convert, the flows column in each output entry will be the *number* of input entries with the same output keys. For example, given this IP_Matrix table:

```
src         dst         pkts  bytes  flows
0.0.0.1     0.0.0.2        3    120      3
0.0.0.1     0.0.0.3        1     40      1
0.0.0.1     0.0.0.4       10    400      5
```

The command "t2_convert IP_Matrix src_IP_Table" yields an IP_Table in which the flows column shows the number of 5-tuple flows with the given source address:

```
src                  pkts  bytes  flows
0.0.0.1                14    560      9
```

but "t2_convert -F IP_Matrix src_IP_Table" yields an IP_Table in which the flows column shows the number of IP pairs with the given source address:

```
src                  pkts  bytes  flows
0.0.0.1                14    560      3
```

Goal: Find the Number of Unique Destination Hosts for Each Source Host

Command line:

```
crl_flow -I -b -Ci=10 if:fxp0 |
  t2_convert Tuple_Table IP_Matrix |
  t2_convert -F IP_Matrix src_IP_Table |
  t2_top -f -n5 IP_Table
```

Sample output:

```
#KEYS            pkts    bytes   flows
# (top 5 sorted by flows)
0.1.0.120        1059    135597  159
0.1.0.5          1224    149885  143
0.4.1.16         188     13280   110
0.4.0.7          1239    140900  87
0.1.0.1          799     67787   78
# end of text table
#KEYS            pkts    bytes   flows
# (top 5 sorted by flows)
0.1.0.120        1159    93601   167
0.1.0.5          1553    119823  157
0.4.0.7          1042    109109  79
0.1.0.1          884     75684   71
0.1.0.65         941     106510  65
# end of text table
```

Explanation:

Remember, since the -F option was used on the second t2_convert, the flows column is actually the number of corresponding entries in the IP_Matrix input table, i.e., the number of IP pairs with the given source address. So, in the first 10 second interval, host 0.1.0.120 sent traffic to 159 different destination hosts, totaling 135597 bytes.

Goal: Find the Top Five Web Servers by Number of Clients

Command line:

```
crl_flow -I -b -Ci=10 -C'filter tcp src port 80' if:fxp0 |
  t2_convert Tuple_Table IP_Matrix |
  t2_convert -F IP_Matrix src_IP_Table |
  t2_top -f -n5 IP_Table
```

Sample output:

```
#KEYS            pkts    bytes   flows
# (top 5 sorted by flows)
0.1.0.52         180     78166   11
0.1.0.108        102     52397   6
0.1.0.62         33      33203   4
0.1.0.182        24      1713    4
0.1.0.231        40      33100   3
# end of text table
#KEYS            pkts    bytes   flows
# (top 5 sorted by flows)
0.1.0.108        192     80733   11
0.1.2.135        66      72221   5
0.1.0.182        20      7135    4
0.3.0.77         131     119220  2
0.1.1.7          145     39893   2
# end of text table
```

Explanation:

This example is similar to the previous one, except that we first filter the traffic to measure only packets sent by HTTP servers. So, in the first 10 second interval, host 0.1.0.52 sent HTTP traffic to 11 different destination hosts, totaling 78166 bytes.

Goal: Find Hosts on Your Internal Network Trying to Spread the CodeRed Worm

Command line:

```
crl_flow -I -b -Ci=60 -C'filter tcp dst port 80 and \
    src net 10.0.0.0/8' if:fxp0 |
  t2_convert Tuple_Table IP_Matrix |
  t2_convert -F IP_Matrix src_IP_Table |
  t2_top -f -n5 IP_Table Sample output:
```

```
#KEYS            pkts    bytes   flows
# (top 5 sorted by flows)
10.0.39.61       7680    460800  7680
10.0.198.103     8960    358400  6144
10.0.39.60       6144    368640  6144
10.0.0.190       7168    299008  4864
10.0.19.1        12544   602112  4608
# end of text table
#KEYS            pkts    bytes   flows
# (top 5 sorted by flows)
10.0.39.61       8448    506880  8448
10.0.39.60       7424    445440  7424
10.0.198.103     7680    307200  6656
10.0.0.141       6144    256000  4352
10.0.213.103     4352    188416  3584
# end of text table
```

Explanation:

Hosts infected with the CodeRed worm try to infect large numbers of other hosts by attempting to open an HTTP connection to random IP addresses (which may or may not actually exist or be running a web server) [20]. With the exception of web caches, most hosts do not open HTTP connections to more

than a few different servers per second, so we should be suspicious of any host that tries to connect to significantly more servers. In particular, hosts infected with CodeRed attempt to open HTTP connections to many tens or hundreds of hosts per second. By using a filter that selects only packets from the internal network (10.0.0.0/8 in this example) to HTTP servers, and by seeing which of the hosts sending those packets are attempting to communicate with the most servers, we produce a list of internal hosts that are behaving suspiciously.

Report Generator

The CoralReef report generator provides a web interface to continuously updated link usage reports. The report generator (t2_report) is a Perl application, using C backends for speed, which receives (via a pipe) either live data or traces taken from crl_flow. t2_report collects and displays timeseries information by using RRDtool [18].

The report generator utilizes many of the features of the CoralReef suite and thus illustrates some of the capabilities of the suite. At configurable intervals (e.g., every five minutes), t2_report produces pie charts and tables of traffic data from the most recent sample interval, and timeseries graphs of data over the last hour, day, week, month, and year. All three report forms present data as bytes, packets, and flows. The pie charts and tables show protocol breakdown, applications, flows, source/destination hosts, unknown TCP and UDP, and source/destination ASes and countries. The timeseries graphs show absolute counts and percentages for protocol breakdown and applications. There are two sets of application timeseries graphs.

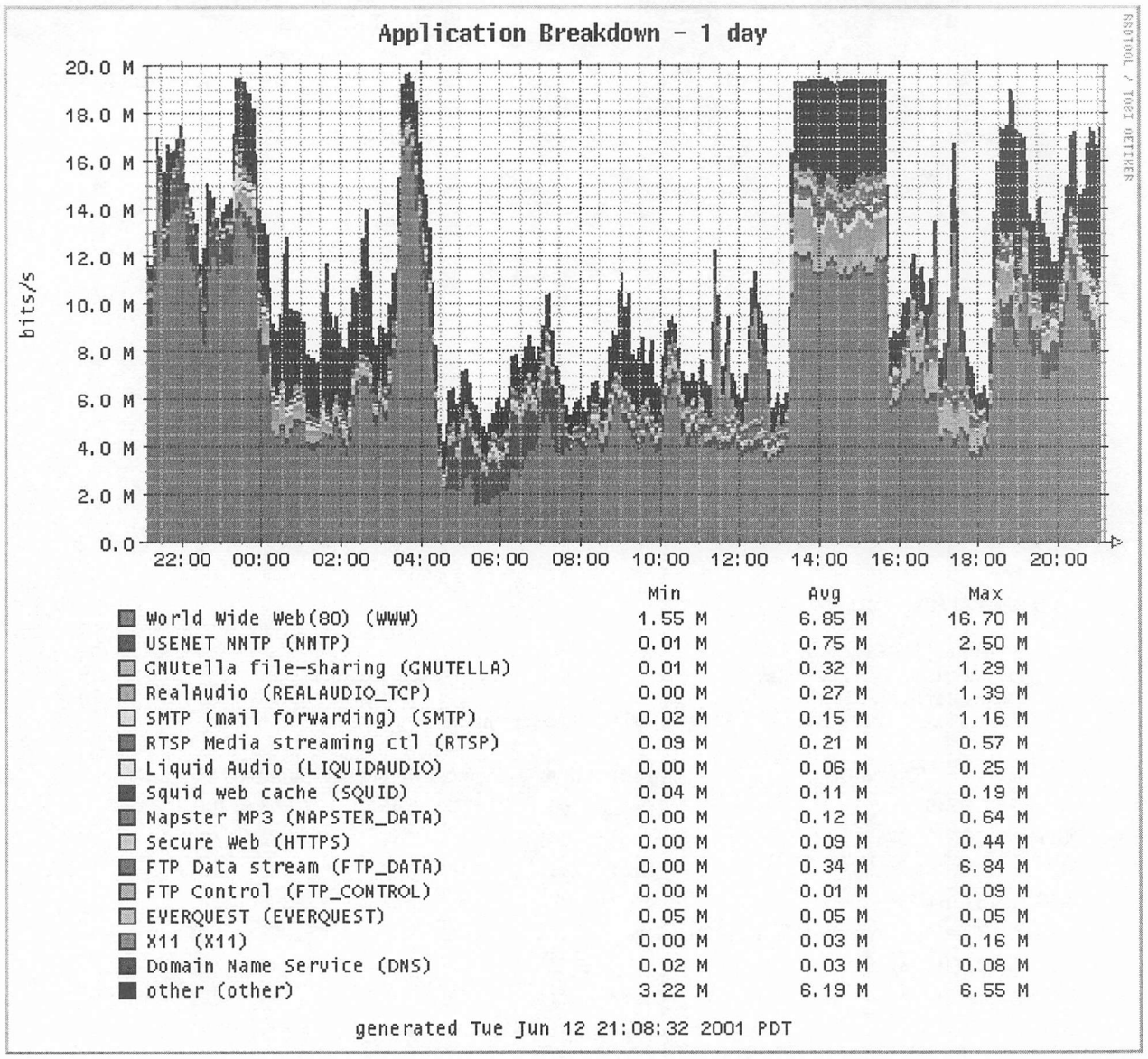
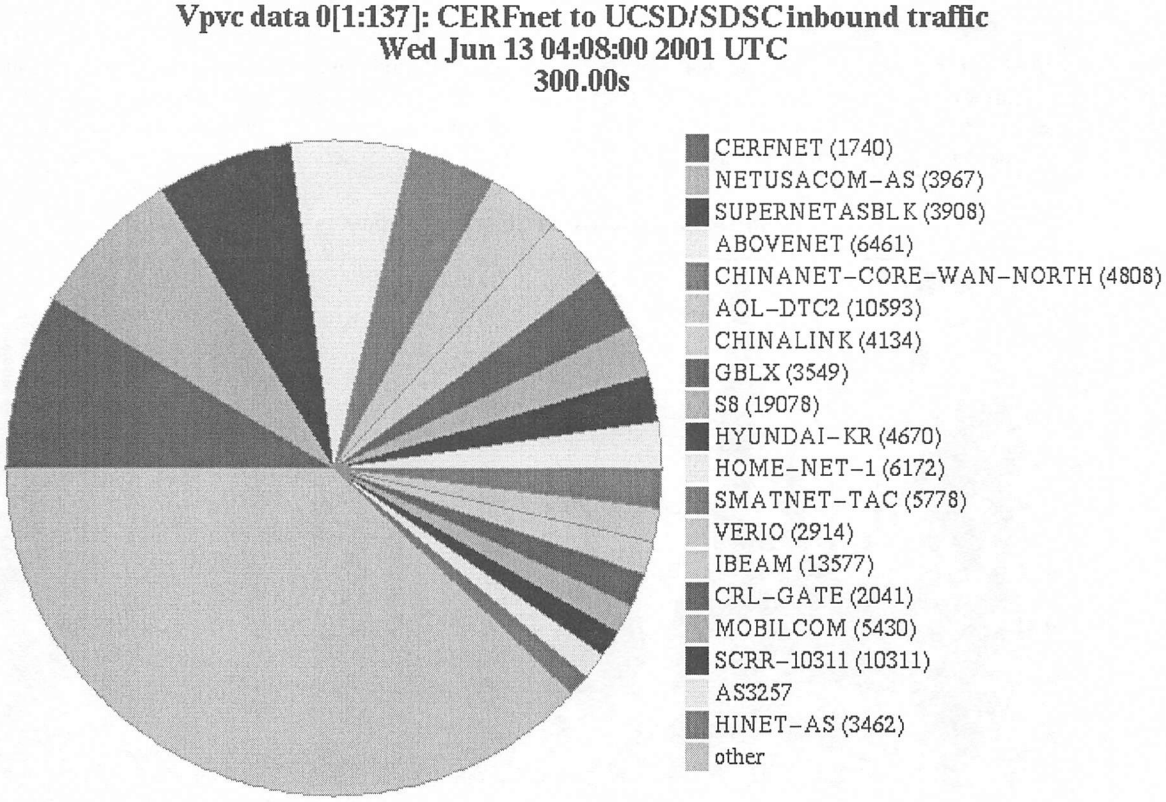


Figure 4: Example of a timeseries plot of application breakdown by bytes.

One shows only the applications specified in the t2_report configuration file, and the other shows the applications with the most traffic in each interval.

To report traffic by AS number, countries, and application names, t2_report must use external data not present in the packets themselves. t2_report uses a library called ASFinder and a routing table, as output by parse_bgp_*, to map IP addresses to AS numbers. To get countries and AS names from AS numbers, t2_report uses NetGeo [14]. Application names are found by the AppPorts library, which uses a prioritized ruleset to map protocol and port numbers to applications. Users can add or modify application rules by editing a simple text file.

Figure 4 shows an example of a timeseries plot of application breakdown by bytes. Figure 5 shows an example of the top source ASes by bytes in a five minute period. The CoralReef web site has a live demonstration of the report generator monitoring the commodity traffic link for the U. C. San Diego.



Overall Performance:

Byte rate: 15.5966 Mbits/s
Packet rate: 4.1023 Kpackets/s
Tuple rate: 163.2733 tuples/s
Total unique Source AS entries: 1789 (top 100 by bytes shown)

Source AS	Mbits/s	% bytes	Kpkts/s	% packets	tuples/s	% tuples
CERFNET (1740)	1.3413	8.60	0.2237	5.45	0.7033	0.43
NETUSACOM-AS (3967)	1.1639	7.46	0.1853	4.52	11.8267	7.24
SUPERNETASBLK (3908)	1.0676	6.85	0.0890	2.17	0.0233	0.01
ABOVENET (6461)	0.9234	5.92	0.1097	2.67	2.3067	1.41
CHINANET-CORE-WAN-NORTH (4808)	0.6644	4.26	0.0940	2.29	3.0333	1.86
AOL-DTC2 (10593)	0.5656	3.63	0.0865	2.11	4.2400	2.60
CHINALINK (4134)	0.5024	3.22	0.0760	1.85	1.6933	1.04
GBLX (3549)	0.4740	3.04	0.0596	1.45	2.8133	1.72

Figure 5: Example of top source ASes by bytes in a five minute period.

Conclusion

CoralReef provides a suite of tools to aid network administrators in monitoring and diagnosing changes in network behavior. CoralReef provides a unified platform to a wide range of capture devices and a collection of tools that can be applied at multiple levels of the network. Its components provide measures on a wide range of real-world network traffic flow applications, including validation and monitoring of hardware performance for saturation and diagnosis of network flow constraints. CoralReef can be used to produce standalone results or produce data for analysis by other programs. CoralReef reporting applications can output in text formats that can be easily manipulated with common UNIX data-reduction utilities (e.g., grep), providing enormous flexibility for customization in an operational setting.

CoralReef provides a balanced collection of features for network administrators seeking to monitor their network and diagnose trouble spots. It serves as a useful bridge between higher level monitoring tools which only work at a coarse level of aggregation and "dump" utilities which may overwhelm the administrator with detail. By covering the range from raw packet capture to real-time HTML report generation, CoralReef provides a viable toolkit for wide range of network administration needs.

Acknowledgments

Support for CoralReef is provided by NSF Grant NCR-9711092, DARPA NGI Contract N66001-98-2-8922, DARPA NMS Grant N66001-01-1-8909, and by CAIDA members. We would like to thank Mike Tesch (formerly CAIDA) and Jambi Ganbar of MCI (formerly CAIDA) for early prototypes and testing; Sue Moon of Sprint Advanced Technology Laboratories and Chris Rapier of Pittsburgh Supercomputing Center for their feedback on CoralReef; Nevil Brownlee, Young Hyun, Colleen Shannon, Daniel J. Plummer, and everyone else at CAIDA for their input and support.

Availability

The CoralReef software package is available for non-commercial use from http://www.caida.org/tools/measurement/coralreef/. Questions about CoralReef can be e-mailed to coral-info@caida.org.

Author Information

David Moore is the Co-Director and a PI of CAIDA (the Cooperative Association for Internet Data Analysis). David's research interests are high speed network monitoring, denial-of-service attacks and infrastructure security, and Internet traffic characterization. His current research includes using the backscatter analysis technique to track and quantify global DoS attacks and Internet worms.

Ken Keys is lead developer of CAIDA's CoralReef project. He has been involved with network research for three years and programming UNIX networking code for over a decade. Ken is known to many for his years of work on TinyFugue, a popular MUD client. Reach him electronically at kkeys@caida.org.

Ryan Koga is CAIDA's resident expert at integrating C and C++ with Perl. He spends most of his time developing libraries for CoralReef and writing assorted programs for other CAIDA projects.

Edouard Lagache is a Researcher and Perl developer with CAIDA. He received his Ph.D. from the University of California, Berkeley in 1995. He has done research on human/computer interaction and social aspects of learning.

kc claffy is Co-Director and a PI of CAIDA, and a resident research scientist based at the University of California's San Diego Supercomputer Center. kc's research interests include Internet workload/performance data collection, analysis and visualization, particularly with respect to commercial ISP collaboration/cooperation and sharing of analysis resources. kc received her Ph.D. in Computer Science from UCSD in 1994.

References

[1] J. Apisdorf, k claffy, K. Thompson, and R. Wilder, "OC3MON: Flexible, Affordable, High-Performance Statistics Collection," *INET'97 Proceedings*, http://www.isoc.org/isoc/whatis/conferences/inet/97/proceedings/F1/F1_2.HTM, June 1997.

[2] Apisdorf, J., k claffy, Kevin Thompson, and Rick Wilder, "OC3MON: Flexible, Affordable, High Performance Statistics Collection," *Proceedings of the 1996 LISA X Conference*, 1996.

[3] Network Associates, "Sniffer home," http://www.sniffer.com.

[4] Brownlee. N., *RFC 2123: Traffic Flow Measurement: Experiences with NeTraMet*, March, 1997.

[5] Brownlee, N., C. Mills, and G. Ruth, *RFC 2722: Traffic Flow Measurement: Architecture*, October, 1999.

[6] Clevertools, "Analyzer – Packet-Sniffer – Network Tools," http://www.clevertools.com/.

[7] Combs, Gerald, et al., "Ethereal – A Network Protocol Analyzer," http://www.ethereal.com/.

[8] Handelman, S., S. Stibler, N. Brownlee, and G. Ruth, *RFC 2724: RTFM: Net Attributes for Traffic Flow Measurement*, October, 1999.

[9] Jacobson, V., C. Leres, and S. McCanne, "tcpdump," Lawrence Berkeley Laboratory, Berkeley, CA, ftp.ee.lbl.gov, June 1989.

[10] KaZaA, "KaZaA media sharing," http://www.kazaa.com/.

[11] Keys, Ken, David Moore, Ryan Koga, Edouard Lagache, Michael Tesch, and k claffy, "The

Architecture of CoralReef: An Internet Traffic Monitoring Software Suite," *PAM2001 – A Workshop on Passive and Active Measurements*, CAIDA, RIPE NCC, http://www.caida.org/outreach/papers/pam2001/coralreef.xml, April, 2001.

[12] Lizcano, P. J., A. Azcorra, J. Solé-Pareta, J. Domingo-Pascual, and M. Alvarez Campana, "MEHARI: A system for Analysing the Use of Internet Services," *Computer Networks*, Vol. 81, pp. 2293-2307, 1999.

[13] McCanne, S., C. Leres, and V. Jacobson, "libpcap," Lawrence Berkeley Laboratory, ftp.ee.lbl.gov, Berkeley, CA.

[14] Moore, David, Ram Periakaruppan, Jim Donohoe, and kc claffy, "Where in the World is netgeo.caida.org?" *INET 2000 Proceedings*, June, 2000.

[15] MRTd, "MRT – multi-threaded routing toolkit," http://www.mrtd.net/.

[16] Narus, "Narus IBI Platform," http://www.narus.com/ibi/.

[17] Niksun, "NetVCR," http://www.niksun.com/products/netvcr.html.

[18] Oetiker, T., *RRDtool – Round Robin Database.*

[19] Waikato Applied Network Dynamics Group, "The DAG Project," http://dag.cs.waikato.ac. nz/.

[20] eEye Digital Security, ".ida 'Code Red' Worm," http://www.eeye.com/html/Research/Advisories/AL20010717.html.

Global Impact Analysis of Dynamic Library Dependencies

Yizhan Sun and Dr. Alva L. Couch – Tufts University

ABSTRACT

Sowhat is an administrative tool that performs global impact analysis of dynamic library dependencies for Solaris systems. Sowhat runs in two phases. It first builds a database of dependencies offline in the background, and then answers user queries and generates reports in real time based upon stored knowledge. Using sowhat, one can find problems with library bindings in large program repositories before these problems annoy potential users.

"I can stands what I can stands, but I can't stands no more!" – Popeye

Introduction

We manage a large Solaris network containing several large shared program repositories, each containing up to 1000 programs apiece. Several times in the last two years we have inadvertently and unpredictably broken user programs and network services during routine upgrades of dynamic libraries. Library dependencies that load libraries from NFS-mounted partitions have also been responsible for random boot-time failures of daemons, because a daemon cannot access its libraries when it needs to load them. Lastly, we can never be sure that a dynamic library is not in use by any program, so we can never delete a dynamic library safely.

In this paper we describe a very simple technique for avoiding such anomalies through "in vivo" global analysis of library dependencies. A Perl script constructs a library dependency database from a user's-eye-view of a system and generates an 'inverted' report, for each dynamic library, of *all* of the programs that will attempt to load it. Using this report, one can easily spot problems such as those mentioned above.

Related work

Our program is related to many other tools that *almost* – but not quite – entirely fail in detecting library dependencies in a heterogeneous and open computing environment. Package managers such as RPM [1] and Depot [2, 8, 9, 11] allow point-of-installation dependency analysis based upon a "closed-world" assumption. As long as either has complete control of the environment, dependencies will likely be satisfied correctly. Combine these with other software installation techniques such as direct compilation, however, and they will fail to spot possible problems in the resulting stew. Neither RPM nor Depot has full knowledge of the user environment in which the software that they install will be utilized.

Change detectors such as tripwire [14] and aide [7] can detect changes to a filesystem but cannot analyze the potential effects. We need to know not only the names of files that changed, but also which programs could be affected by libraries that have changed, and even perhaps which programs might be broken by a particular 'make install'.

Pre-existing tools most applicable to our problem act to control the user's environment carefully so that conflicts should not occur. The Soft [5] environment control system manages library bindings by careful control of the user's environment variables. Soft was preceded by much other work on creating software modules that can be invoked by users on demand [4, 6]. These inspired our own software module mechanism that sowhat understands and analyses.

Others have faced the same problems with libraries and opted to control the dynamic linking environment carefully in order to avoid the need for approaches like sowhat. Vendor-supplied software has been hampered in Linux by the large number of differing distributions of what is essentially the same core operating system. Differences in distributions can often break software, so that a product that works properly in one distribution may not work in another.

The Linux Standard Base (LSB) project [13] seeks to provide a dynamic linking environment within Linux in which vendor-provided software is guaranteed to execute properly. The goal of LSB is to identify a set of core standards that must be shared among distributions in order to guarantee that a product that works properly in one of them will work in all compliant distributions. These standards include requirements for the content of dynamic libraries, as well as standards for locations of system files used by library functions.

With these standards in hand, the LSB provides tools with which one can certify both environments and programs to be compliant with the standard. Linux distributions can be examined by an automatic certification utility that checks link order, versions of libraries, and locations of relevant system files. A distribution may have more libraries than the standard specifies, but the libraries specified in the standard must be first to be scanned during linking and must

contain the appropriate versions of library subroutines. Another certification utility checks that the binary code for linux applications only calls library functions protected by the standard. Since the LSB tools solely analyze the contents of binary files, they can check closed-source executables for compliance.

The LSB's library version probing is a much deeper library analysis than our tool performs, though not beyond the scope of future work. For example, sowhat relies upon the names of libraries to indicate versions, while LSB scans them for embedded version strings, so that it can accurately determine the content and versions of renamed or even misnamed libraries.

Dynamic Libraries and Global Analysis

The cause of our problems is that in a modern UNIX environment, the file containing an executable program is seldom the only component required in order to execute the program. Each executable binary file contains references to one or more dynamic libraries that are linked into the program after it is invoked and before it begins execution (through the dynamic linker ld.so). The typical reason for this way of segmenting programs is to share runtime memory; one in-memory copy of a library may be shared among several executables running at the same time. But if the libraries needed at runtime by a specific program are changed or deleted, the referring program may change in behavior or fail to run at all.

While it is possible (through the command ldd) to examine which libraries are loaded by any specific program, no general mechanism except sowhat exists for examining which programs load a specific library. This is because while the former question involves examining one executable, the latter involves examining *all possible executables* that could load the library. The former question is *local* in scope, while the latter is *global*. In practice, this means that without a tool such as sowhat, one can never safely delete any library in the system without the risk of breaking an unknown program. If user uptime is more important than disk space, this means one can never delete or upgrade any library at all because the impact of such a change is unknown. One can only add libraries. This leads to 'library rot' much like 'filesystem rot' [3], in which libraries gradually fill up with useless files that one cannot delete with any assurance of lack of impact.

Analyzing Dependencies

The basic function of sowhat is very straightforward. One runs it as a normal user. Sowhat parses the user's PATH environment variable to create a list of programs to scan. For each program, it parses the output of the Solaris utility 'ldd' [12] to generate an index of the libraries the program will load. It inverts this index so that it can list programs that call each library and stores the inverted index in a database from which it can generate reports. One can easily limit the report to specific libraries of interest by listing them on the command line.

How *sowhat* Works

Sowhat is currently written to analyze a Solaris 2.x environment. In this environment, the dynamic linker ld.so utilizes hard-coded library paths encoded into the executable program, as well as search requests that tell ld.so to scan a library path for a library matching a given name pattern and version. This scan checks all directories listed in the environment variable LD_LIBRARY_PATH. The function of ld.so is mimicked by the diagnostic program ldd, which reports the full pathnames of libraries that will satisfy each search request when ld.so is invoked prior to program execution.

To function properly, sowhat has to intimately understand the possible responses of the ldd command. The command ldd lists dynamic dependencies of executable files or shared objects. For example, for the executable file /local/bin/g++, ldd lists the path names of all shared objects that will be loaded whenever /local/bin/g++ is loaded, e.g.:

```
libc.so.1 =>  /usr/lib/libc.so.1
libdl.so.1 => /usr/lib/libdl.so.1
```

These records are *relative*; g++ asks for the first version of the library in the current library path matching the pattern libdl.so.1. This matches /usr/lib/libdl.so.1 .

The output of ldd can look quite different for vendor-supplied software. Consider the output of ldd for the tnshut command supplied with Sun's TotalNet software:

```
libdl.so.1 =>     /usr/lib/libdl.so.1
libsocket.so.1 =>/usr/lib/libsocket.so.1
libnsl.so.1 =>    /usr/lib/libnsl.so.1
libintl.so.1 =>   /usr/lib/libintl.so.1
libc.so.1 =>      /usr/lib/libc.so.1
libmp.so.2 =>     /usr/lib/libmp.so.2
/usr/platform/SUNW,
        Ultra-250/lib/libc_psr.so.1
```

The last library has an *absolute* binding. It must exist in exactly that place in the filesystem or the program will not function. In this case there is a good reason for the absolute binding as the existence of the library in question is dependent upon the sub-architecture of the particular machine. If it is not present, the system in question has the wrong architecture to run the software!

In order to find failures, sowhat must also understand the meaning of the various and sundry error messages provided by ldd. These include:

1. No match to library pattern:

```
libucb.so.1 => (file not found)
```

2. Correct version not found:

```
libm.so.1 (SUNW_1.1) =>
        (version not found)
```

In both cases, sowhat will record the actual library name as NotHere. If you then ask sowhat to list the

programs that use the library NotHere, it will list all programs that cannot run due to missing libraries.

Environmental Awareness

Alas, any such script must be cognizant of the detailed structure of the user's environment – a matter of local operating policy. In particular, sowhat must be able to reconstruct any environment available to a user in order to test for problems within each one.

At Tufts EECS, we utilize a simple derivative of software module management [4, 5, 6] to allow users to dynamically add modules to their environment, modifying both their PATH and LD_LIBRARY_PATH as needed. Users need only type the shell command:

```
use packname
```

(where packname is the name of the package) in order to modify their environments for a specific package. We use this mechanism to allow users access to a variety of software that is not accessible except by specific request, such as vendor software for computer-aided design.

The 'use' command above invokes a package-specific startup script to modify the user's environment appropriately so that the desired package will work properly. For example, use new executes /local/env/new.cshrc:

```
set path = ($path[1-3] \
  /local/new/bin $path[4-])
setenv MANPATH \
  /local/new/man:${MANPATH}
setenv LD_LIBRARY_PATH \
  /local/new/lib:${LD_LIBRARY_PATH}
```

which has the effect of including the beta software testing tree /local/new in the user's PATH, MANPATH, and LD_LIBRARY_PATH.

The way the use command works is quite straightforward. The tcsh alias:

```
alias use \
  'set packages = ( \!* ) ; \
   source /local/lib/use'
```

sources the script /local/lib/use with $packages set to the appropriate package name:

```
#! /usr/bin/tcsh
if ($?packages) then
  foreach pkg ($packages)
    if (-f /local/env/$pkg.cshrc) then
      echo Using package $pkg : \
        setting up your environment
      source /local/env/$pkg.cshrc
    else
      echo There is no package $pkg : \
        please check your spelling.
    endif
  end
endif
unset packages
```

This script in turn sources a setup script from /local/env (starting with the package name) that sets PATH and LD_LIBRARY_PATH appropriately for the new module.

Analyzing Customizations

Using packages of this kind provides not only a way to avoid user confusion about commands most users do not need; it also provides a marvelous hiding place for library binding errors. To find library problems, one must analyze each package environment that the user can construct. Starting from each user's default environment, *Sowhat* constructs each custom environment available to users, one by one, and then analyzes effects of any additional libraries or executables. It does not test the effect of executing more than one 'use' at a time, though this would be helpful if not ridiculously time-consuming.

Results

Analyzing all packages available in a large system is a very time-consuming process. It takes between 10 and 20 minutes to analyze the configuration of a typical user on a Sun Enterprise-250 server, depending upon system load. Typically we invoke *sowhat* for data collection in background or overnight runs and store the results for later perusal and comparison. Runs of sowhat can be incremental or restarted from a previous failure. Sowhat is also capable of running in a differential mode in which it compares its recorded data against the system to detect potentially damaging changes.

Sowhat has educated us about our practices and the state of our program repositories in a way we could never have seen without it. It provides a previously unavailable window into our systems that informs us not only of potential problems, but also gives us a general overview of the health of our program repositories and the impact of our management practices.

Observed Problems

It was remarkable to us just how many things were wrong with our repositories. Using sowhat we detected the following kinds of problems:

1. Binary program invalid.
   a. Wrong subarchitecture.
   b. Wrong exec format.
2. Missing library.
   a. Nonexistent library.
      i. Due to absolute library pathname.
      ii. In all library path members.
   b. Incompatible library version.
      i. Due to absolute library pathname.
      ii. In all library path members.
   c. Incompatible library subarchitecture.
      i. Due to absolute library pathname.
      ii. In all library path members.

Rarely, the program we analyzed was not a Solaris program at all. Unbelievably, we found several Linux x86 programs installed in our Solaris repository!

A more subtle and serious error was that several programs in the correct exec format were compiled for

a different hardware subarchitecture, e.g., 64-bit code on a 32-bit machine. These crept into our repository due to addition of 64-bit servers, while no one noticed that they could not be used on most of the workstations.

The remaining errors we found are the ones we were looking for. Several programs had outlasted their libraries by several years; we deleted libraries because we were completely unaware of the program's need for them. Other programs were compiled to hard-coded library locations, and these locations had been moved by operating system updates. Still others needed an older or newer version of the library than was available. Finally, some programs were made available on a machine having an inappropriate subarchitecture, which showed up in sowhat as an incompatible *library* subarchitecture.

Avoiding errors

Sowhat is very useful for analyzing the results of messy repository maintenance, but is equally useful in preventing the messes before they happen. Perhaps the nicest thing about sowhat is that one can ask it about the future impact of any change upon the user environment.

If one wishes to change or delete a library, sowhat can tell which programs will change in function or break based upon this change. One can then test these programs after the change to insure that they still work appropriately. If there are no such programs then the library may be deleted with no impact upon users.

If one wishes to delete a program, sowhat will suggest libraries that can be deleted along with it. These are the libraries that only the doomed program uses. So libraries never need to be kept around 'just in case' some program uses them. This greatly simplifies maintenance of repositories because they no longer need fill up with libraries that no program uses, just because it is unsafe to delete them without knowing which programs do.

Sowhat's differential mode not only notifies one of the effects of intentional library replacement, but also the effects of unintended or malicious changes. Unlike Tripwire [14] and Aide [7] it can detect not only a malicious change, but also identify its potential sphere of effect.

After operating system upgrades, sowhat can tell you which library bindings changed for which programs. This allows you to test those programs for possible problems created by the upgrade. One can also run it in 'differential mode' to compare the *user environments* on two hosts sharing the same command repository.

When we first ran sowhat, on one of our machines named andante, out of 9780 executables, we found 12 programs with missing libraries. Out of 61 packages, eight packages did not work for a variety of reasons. Some of the numbers generated by sowhat are a bit staggering: if we change /usr/lib/libc.so.1, 2237 executables will be affected!

By running sowhat on several different machines, we can determine the differences and inconsistencies in their user environments. For example, we discovered that libm.so.1(SUNW_1.1) is missing on andante, but present on other machines. On some machines we observed "execution failed" or "Exec format error" messages that were not seen on others. This is due to sub-architecture differences between the machines.

Lessons Learned

The main lesson that we learned from sowhat is that one cannot judge the impact of changing a dynamic library without some form of global analysis. Our tool does this analysis sufficiently well that one can rather accurately predict the sphere of possible effects of any potential change. Before we started using sowhat, we had already experienced several service failures due to library replacements, notably libz.so, which is used by a surprising variety of open source tools. This kind of potential effect was invisible to us before we wrote *sowhat*.

Alas, sowhat has several limitations. The first is that to analyze the user environment, sowhat must also know how that environment can change based upon user needs. This is a site-specific system property. In turn, to perform a complete analysis on a given site, sowhat must be updated to understand the mutations that can occur in the user environment at that site.

By far, the largest blind spot in sowhat is that it does not detect *conflicts* between user-invoked packages. It is quite possible that, by a specific sequence of environmental modifications, a user can produce a broken environment. The environment-modifying scripts can in principle do *anything*. There is no guarantee that one will not undo the good of another, and it is impractical to check all sequences of executions of these scripts. This is not a limit of sowhat, but one imposed by our environment and operating policy.

Future Work

Sowhat is a fairly closed-ended tool with a specific function that it performs quite well. We are unlikely to expand upon its basic functionality. However, we have already been begged to port this utility to Linux and this port is likely to become available in the near future. Other architectures are less likely as porting targets.

Several open questions may be attacked by future tools with differing scopes. It would be nice, e.g., to automate the process of comparing user environments and checking for homogeneity between various machines. This would be especially useful if it operated also in a *heterogeneous* environment, e.g., comparing commands and versions available to Solaris and Linux users.

Availability

Sowhat is freely available from http://www.eecs.tufts.edu/~couch/sowhat. It is a Perl script that requires access to a MySql database through the DBI and DBD::Mysql interfaces in order to record its results.

Acknowledgements

Many people contributed to the ideas in this project. The original idea came from a conversation with Steve Moshier about the difficulty of uninstalling software in a UNIX environment. Andy Davidoff and Michael Gilfix helped to define the problem, offered valuable insights, and provided valuable comments on the paper as it was being written.

Author Biographies

Alva L. Couch was born in Winston-Salem, North Carolina where he attended the North Carolina School of the Arts as a high school major in bassoon and contrabassoon performance. He received an S.B. in Architecture from M. I. T. in 1978, after which he worked for four years as a systems analyst and administrator at Harvard Medical School. Returning to school, he received an M.S. in Mathematics from Tufts in 1987, and a Ph.D. in Mathematics from Tufts in 1988. He became a member of the faculty of Tufts Department of Computer Science in the fall of 1988, and is currently an Associate Professor of Electrical Engineering and Computer Science at Tufts. Prof. Couch is the author of several software systems for visualization and system administration, including Seecube (1987), Seeplex (1990), Slink (1996) Distr (1997), and Babble (2000). In 1996 he also received the Leibner Award for excellence in teaching and advising from Tufts. He has assisted in maintaining the Tufts computer systems for Computer Science teaching and research since 1985, when he was a Ph.D. student. He can be reached by surface mail at the Department of Electrical Engineering and Computer Science, 161 College Avenue, Tufts University, Medford, MA 02155. He can be reached via electronic mail as couch@eecs.tufts.edu. His work phone is (617)627-3674.

Yizhan Sun is a Masters student at Tufts University who expects to graduate in December of 2001. She also holds an MS in Physics from Boston College. Aside from being a teaching assistant at Tufts, she was a system administrator in the Center for Connected Learning at Tufts from May 2000 to Aug 2000. In her spare time, she enjoys reading and swimming. She can be reached by email to ysun@eecs.tufts.edu, or by postal mail to 455 Boston TPKE Apt 4, Shrewsbury, MA, 01545.

References

[1] Bailey, E., *Maximum RPM*, Red Hat Press, 1997.

[2] Colyer, Wallace, and Walter Wong, "Depot: a Tool for Managing Software Environments," *Proc. LISA-VI*, Usenix Assoc., 1992.

[3] Couch, A., "SLINK: Simple, Effective Filesystem Maintenance Abstractions for Community-Based Administration," *Proc. Lisa-X*, Usenix Assoc, 1996.

[4] Elling, R., and M. Long, "Usersetup: A System for Custom Configuration of User Environments, or Helping Users Help Themselves," *Proc. LISA-VI*, Usenix Assoc., 1992.

[5] Evard, R. and R. Leslie, "Soft: A Software Environment Abstraction Mechanism" *Proc. LISA-VIII*, Usenix Assoc., 1994.

[6] Furlani, J. L., "Modules: Providing a Flexible User Environment," *Proc. LISA-VI*, Usenix Assoc., 1992.

[7] Lehti, Rama, "AIDE – Advanced Intrusion Detection Environment," http://www.cs.tut.fi/~rammer/aide.html.

[8] Manheimer, Kenneth, Barry Warsaw, Stephen Clark, and Walter Rowe, "The Depot: A Framework for Sharing Software Installation Across Organizational and UNIX Platform Boundaries," *Proc. LISA-IV*, Usenix Assoc., 1990.

[9] Rouillard, John P., and Richard B. Martin, "Depot-Lite: A Mechanism for Managing Software," *Proc. LISA-VIII*, Usenix Assoc., 1994.

[10] Sellens, John, "Software Maintenance in a Campus Environment: The Xhier Approach," *Proc. LISA-V*, Usenix Assoc., 1991.

[11] Wong, Walter C., "Local Disk Depot – Customizing the Software Environment," *Proc. LISA-VII*, Usenix Assoc., 1993.

[12] LDD man page, "man ldd," Sun Microsystems Inc.

[13] The Linux Standard Base Project, "The Linux Standard Base," http://www.linuxbase.org.

[14] Tripwire, Inc, "The Tripwire Security Scanner," http://www.tripwire.com.

Tools to Administer Domain and Type Enforcement

Serge Hallyn and Phil Kearns – College of William and Mary

ABSTRACT

Domain and Type Enforcement (DTE) is a mechanism that can be effective in providing extremely fine-grained mandatory access control in complex networked systems. Unfortunately, this level of control comes at a price: the configuration of a DTE policy is usually in the form of a dense ASCII file which does not lend itself to an understanding or administration of the policy itself. We describe a set of graphical tools which aid in that understanding and administration.

Introduction

Precise access control is generally acknowledged as a means of providing greater system security. The gross access control of the standard Unix scheme of twelve protection bits specifying access for the owner of a binary, the owner's group, and the rest of the world has lead to two points of vulnerability in modern Unix systems: the existence of "set user id" (root) binaries and the root account being exempt from all access restrictions. Responses to these vulnerabilities include techniques such as access control lists (Linux-ACL [8] is a recent development in access control lists) and capabilities (such as POSIX capabilities [7] as partially implemented in the Linux v2.2 kernel). Each technique attempts to allow finer-grained access control than the standard Unix mechanism, especially for processes running with root privilege. For example, using POSIX capabilities, the `talkd` service may only need access to restricted network ports, so that it may be started with only the `CAP_NET_BIND_SERVICE` capability. If `talkd` is later compromised, the attacker's privileges on the system are still very limited, despite being root on the system.

Type Enforcement was introduced by Boebert and Kain [4] in 1985 as a method of implementing integrity systems without relying on a trusted user. It labeled objects as well as subjects, and specified access from subjects to objects and subjects to subjects in two matrices. Subject labels were called domains, and object labels were called types. Subjects to object access could be read, write, and execute. Type Enforcement was implemented first in the Secure Ada project (LOCK) [10] , and later by TIS in Trusted XENIX [1] . Secure Computing still uses TE in its Sidewinder firewall product [6] .

Domain and Type Enforcement was first presented by O'Brien and Rogers [10] and is an extension of TE. It differs mainly in specifying policies in an intuitive policy language rather than using two matrices. Domain-to-domain transitions are allowed by the execution of special binaries designated as entry points to the target domain. TIS did the first Unix implementation of DTE [3] on a BSD system.

Access control techniques similar to TE and DTE are a continuing source of research. Most notably, NSA's Security-Enhanced Linux project [2] uses a mechanism much like TE and DTE to control the access rights for "process labels" (similar to domains).

The Problem

We have implemented DTE for Linux 2.3 and 2.4. Administering our implementation of DTE consists of editing a plaintext policy file. The system must be rebooted to effect the changes. The policy file consists of several sections. The first section enumerates the types and domains. Next are specified the default type for the file system root ("/") and its children, and the domain in which to run the first process. This is followed by a detailed definition of each domain. For each domain, we specify the entry points, permitted type access, permitted domain transitions, and permitted signals to processes in other domains. The last section lists the type assignment rules. See the sample policy file in Listing 1.

This policy file should be viewed as part of a much larger file which defines the DTE policy for a networked Unix system. It was written primarily to demonstrate how DTE could defend against root compromise via the recent `wu-ftpd` exploit [5]. The ftp daemon provided with Red Hat Linux 6.2 contains a well-known string format vulnerability that allows any remote (or local) user to obtain a root shell. This policy file was implemented in a version of Linux 2.3 which had been modified to support DTE [9] and was shown to eliminate root compromise through the `wu-ftpd` vulnerability.[1]

The policy defined in Listing 1 attempts to confine the access rights of the ftp daemon (/usr/sbin /in.ftpd) so that a successful overflow exploit against it does not compromise the rest of the system – certainly, we do not want to allow the attacker to run a shell with root privileges. Lines 19-21 of the policy define the ftp_d domain in which the daemon is required to execute. Note that processes in the ftp_d

[1] See the cited paper for details of the syntax.

domain may only execute binaries beneath /lib and /home/ftp/bin, in addition to /usr/sbin/in.ftpd itself. This follows from the rules

```
rxd->lib_t
rxd->ftp_xt
```

on line 20, together with the type assignments on lines 29, 30, and 34. If there is, in fact, an exploitable overflow in the ftp daemon, our policy will ensure that the daemon cannot execute a shell, which is assumed not to reside in a location accessible for execution to /usr/sbin/in.ftpd.

The ability to use DTE to confine the access rights of an important process like the ftp daemon, to which we want to allow relatively easy network access, clearly limits the damage that can be done if that daemon is compromised. This power comes at an obvious price: the DTE policy file is dense, relatively unstructured, text. In some sense, this is a natural consequence of fine-grained access control. We simply have a lot to specify when we specify the DTE policy for a complex system with many domains and types. However, the impact of an error in this file may be disastrous. We address this problem with a tool that takes the DTE policy file as input and produces a plugin for a Perl/Tk graphical analysis tool.

Our Solution: DTEedit/DTEview

DTEedit is a DTE policy file editor that understands, and enforces, proper syntax of the policy file. DTEedit produces a well-formed policy file and (in a separate file) a representation of the policy expressed as Perl code. DTEview is a Perl/Tk program that assists in the administration of DTE. Its representation of the DTE policy is provided by the Perl plugin output by DTEedit. DTEview internally treats the policy as a directed graph in which nodes represent types and domains. An edge from a domain to a type is labeled to indicate the appropriate access rights; an edge from a domain to a domain indicates a possible domain transition through an entry point or, although not relevant in this example, signals allowed from one domain to another. DTEview takes various closures of the

```
01 # ftpd protection policy
02 types root_t login_t user_t spool_t binary_t lib_t passwd_t shadow_t dev_t \
03       config_t ftpd_t ftpd_xt w_t
04 domains root_d login_d user_d ftpd_d
05 default_d root_d
06 default_et root_t
07 default_ut root_t
08 default_rt root_t
09 spec_domain root_d (/bin/bash /sbin/init /bin/su) (rwxcd->root_t rwxcd->spool_t \
10     rwcdx->user_t rwdc->ftpd_t rxd->lib_t rxd->binary_t rwxcd->passwd_t \
11     rxwcd->shadow_t rwxcd->dev_t rwxcd->config_t rwxcd->w_t) (auto->login_d \
12     auto->ftpd_d) (0->0)
13 spec_domain login_d (/bin/login /bin/login.dte) (rxd->root_t rwxcd->spool_t \
14     rxd->lib_t rxd->binary_t rwxcd->passwd_t rxwcd->shadow_t rwxcd->dev_t \
15     rxwd->config_t rwxcd->w_t) (exec->root_d exec->user_d) (14->0 17->0)
16 spec_domain user_d (/bin/bash /bin/tcsh) (rwxcd->user_t rwxd->root_t \
17     rwxcd->spool_t rxd->lib_t rxd->binary_t rwxcd->passwd_t rxwcd->shadow_t \
18     rwxcd->dev_t rxd->config_t rwxcd->w_t) (exec->root_d) (14->0 17->0)
19 spec_domain ftpd_d (/usr/sbin/in.ftpd) (rwcd->ftpd_t rd->user_t rd->root_t \
20     rxd->lib_t r->passwd_t r->shadow_t rwcd->dev_t rd->config_t rxd->ftpd_xt \
21     rwcd->w_t d->spool_t) () (14->root_d 17->root_d)
22 assign -u /home user_t
23 assign -u /tmp spool_t
24 assign -u /var spool_t
25 assign -u /dev dev_t
26 assign -u /scratch user_t
27 assign -r /usr/src/linux user_t
28 assign -u /usr/sbin binary_t
29 assign -e /usr/sbin/in.ftpd ftpd_xt
30 assign -r /home/ftp/bin ftpd_xt
31 assign -e /var/run/ftp.pids-all ftpd_t
32 assign -r /home/ftp ftpd_t
33 assign -e /var/log/xferlog ftpd_t
34 assign -r /lib lib_t
35 assign -e /etc/passwd passwd_t
36 assign -e /etc/shadow shadow_t
37 assign -e /var/log/wtmp w_t
38 assign -e /var/run/utmp w_t
39 assign -u /etc config_t
```

Listing 1: Sample policy file.

graph representation of the policy in response to user queries. In this paper we restrict the description of DTEview to those features needed to detect a problem with the sample policy file listed above, although clearly the tools will analyze any DTE policy.

We begin by showing a DTEview file type analysis window in Figure 1. We start a directory traversal at /home/ftp. This results in a display of the contents of the /home/ftp directory, which in this case are the bin/ and incoming/ subdirectories. Each element of the display is labeled with its DTE type. By left clicking on the arrows pointing out of the subdirectory elements, one may traverse the hierarchy as deeply as needed, seeing associated types for all file system objects. The most relevant parts of this display for our purposes are that:

- the directory subtree rooted at /home/ftp/bin is of type ftpd_xt, and hence, the ftp daemon can execute any binary beneath /home/ftp/bin, and
- the /home/ftp directory is of type ftpd_t.

The next DTEview window is the process tree analyzer shown in Figure 2. When invoked, the tool starts with the domain of /sbin/init and shows the one-step domain transitions possible by the entry point mechanism. In our example, the top two rows of rectangles are drawn. Each rectangle represents a process capable of running in the system, labeled with the domain in which it runs. Left clicking on a rectangle in the second row will show, in similar form, domains into which a process represented by the rectangle may transition through an entry point. No such transitions are shown in Figure 2. Right clicking on a rectangle shows the types to which the process/domain has access rights. The types are represented by the circles on the third row of Figure 2; this row was obtained by right clicking on the rectangle associated with the ftp daemon, and filtering the display to show only types to which the process has directory write (c) access.

Note that the ftp daemon, running in the ftpd_d domain, has "rwcd" access rights to any files or directories of type ftpd_t. Clicking on the ftpd_t icon now brings up the window shown in Figure 3. We see in the last two lines that ftpd_t is assigned to /home/ftp and all its children. This is significant because the directory write access to /home/ftp means that ftpd_d can rename and replace /home/ftp/bin, under which it has execute access. If the daemon is susceptible to an overflow-based exploit, for example, it is possible to replace the /home/ftp/bin directory with one populated with Trojan Horse binaries. Since the /home/ftp/bin directory is intended to contain service binaries, such as ls, for the ftp daemon, it is easy to imagine a replacement ls which replies with the contents of /etc/passwd and /etc/shadow to enable password cracking off-line. Although this is not as immediately serious as a root shell, it still represents a substantial security threat for a real system.

The above problem is an error pattern: if a domain can change binaries or directories which contain binaries to which the domain has execute access rights, then processes in that domain cannot have their

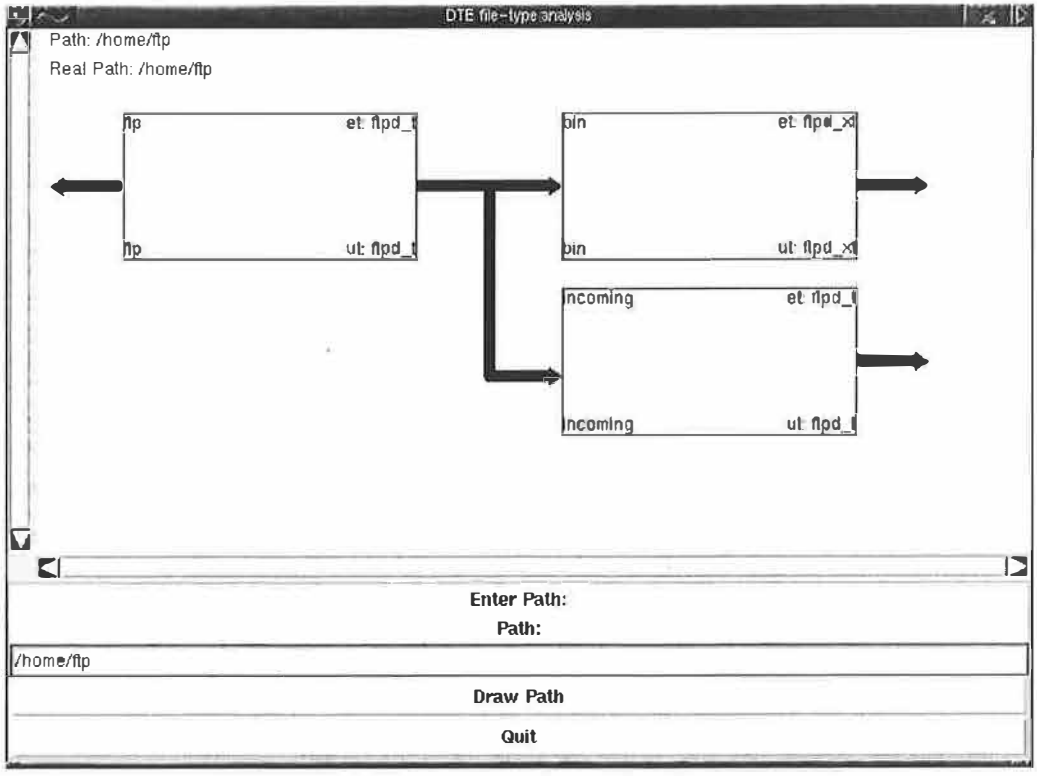


Figure 1: File type analyzer window for sample policy.

execution behavior effectively constrained. When DTEview is started, it runs a check for such a situation for all domains listed in a control file. If the error pattern is found, a popup window notifies the user. Figure 4 shows the popup window that appears when DTEview is applied to the sample policy.

Reachability

Treating the DTE policy as a graph allows us to make queries concerning access by domains to types and domains. Restricted to looking at the policy file, even a simple case such as asking whether a domain has create access to a directory can become tedious.

The task is complicated significantly when we consider that a domain may be able to access a file after performing a domain switch. For instance, a compromised web server may be allowed to write but not execute /bin/sh. If it can subsequently switch to a user domain, for instance, to run a CGI script, and execute what it wrote from the user domain, this could be just as bad. DTEview can answer the question "Can a process under domain daemon_d write /bin/sh within two domain transitions?"

In order to aid a system administrator in analyzing query results, one can specify labels on domain transitions. For instance, the security group may have written a login daemon that uses Java Rings [11] for authentication, and which they have verified to be correct. They can then label domain transitions out of login_d, provided it was entered through /bin/login.javaring, with a string indicating the user's identity has been verified by a trusted program. In a subsequent query as to whether a web server is able to enter the root_d domain, the security administrator can ignore domain transition paths which contain such a transition, as this transition acts as a barrier to any unauthorized users.

Using these features of DTEview, administrators can begin to verify specific assertions concerning the policies which they are writing.

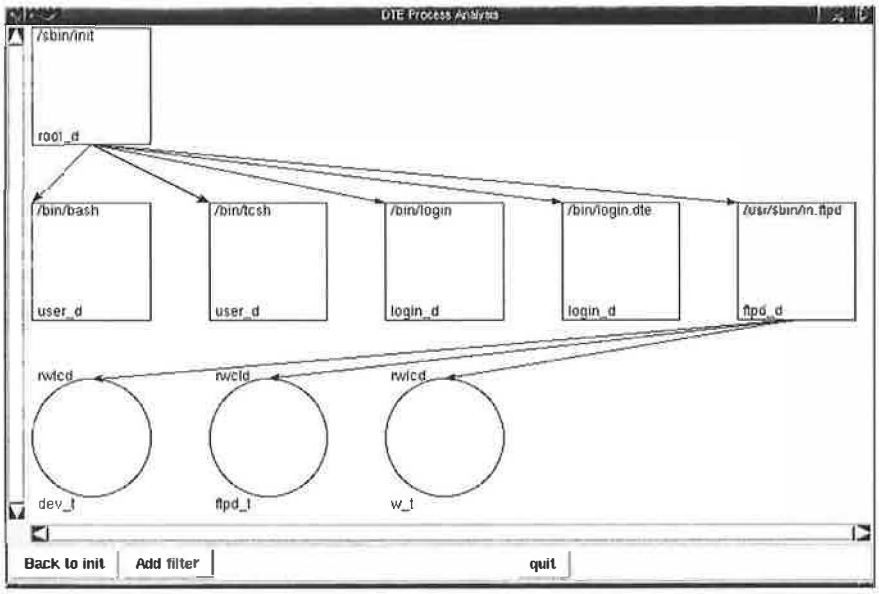


Figure 2: Process tree analyzer window for sample policy.

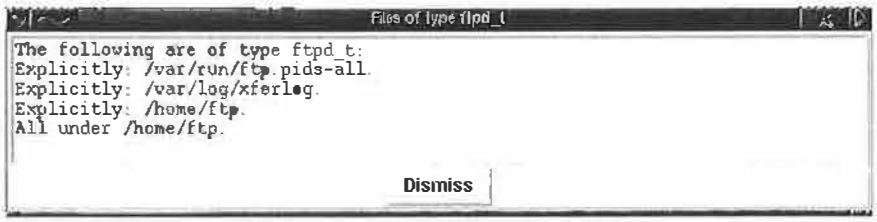


Figure 3: Window showing type assigns for ftpd_t.

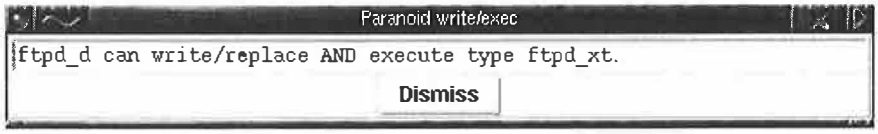


Figure 4: Popup window.

Concluding Remarks

Fine-grained access control, as provided by DTE, has obvious advantages in securing a modern networked system. It also has the obvious disadvantage of requiring a detailed and lengthy specification of the access control policy. An obscure text-based configuration file does not lend itself to understanding or debugging an access control policy. We have given an overview (by example) of a technique which models the policy in graph-theoretic form providing a means for automated analysis and graphical display of important aspects of the policy. Equally important is the fact that the model also provides a formal framework within which we can state and prove theorems about the logic behind the analysis and displays presented by DTEview.

Future Work

DTEedit and DTEview are a good start at easing security policy administration. Work is also under way to aid in constructing whole policies from several coherent pieces (modules), to offer still more intuitive ways of representing policies for analysis, and to provide more powerful means for proving policy properties. Naturally, this is in parallel to the continued implementation and maintenance of DTE itself, which is currently being ported to work with the Linux Security Module [12] project.

Availability

DTEedit and DTEview are available under http://www.cs.wm.edu/~hallyn/dte.

Thanks

The authors wish to thank the the USENIX organization, as well as the anonymous reviewers and Adam S. Moskowitz for helpful comments.

Author Information

Serge Hallyn <hallyn@cs.wm.edu> is a Ph.D. candidate at the College of William and Mary, whose research concerns systems security.

Phil Kearns <kearns@cs.wm.edu> is on the faculty of the Department of Computer Science at the College of William and Mary. His current research interests include distributed systems and operating systems.

References

[1] National Security Agency, "Evaluated Platforms: Trusted Xenix," http://www.radium.ncsc.mil/tpep/epl/entries/CSC-EPL-92-001-A.html.

[2] National Security Agency, "Security-enhanced Linux," http://www.nsa.gov/selinux, 2000.

[3] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat, "A Domain and Type Enforcement Unix Prototype," *Proceedings of the Fifth USENIX UNIX Security Symposium*, June, 1995.

[4] Boebert, W. E. and R. Y. Kain, "A Practical Alternative to Hierarchical Integrity Policies," *Proceedings of the National Computer Security Conference*, Vol. 8, Num. 18, 1985.

[5] CERT, "Two Input Validation Problems in ftpd," http://www.cert.org/advisories/CA-2000-13.html, July, 2000.

[6] Secure Computing, "Type Enforcement Technology for Access Gateways and VPNs," http://www.secure-computing.com.

[7] POSIX Security Working Group, *POSIX System API Amendment 1003.1e: Protection, Audit, and Control Interfaces* (withdrawn), October, 1997.

[8] Andreas Grunbacher, http://acl.bestbits.at.

[9] Hallyn, Serge and Phil Kearns, "Domain and Type Enforcement for Linux," *Proceedings of the Atlanta Linux Showcase*, 4, October, 2000.

[10] O'Brien, R. and C. Rogers, "Developing Applications on Lock," *Proceedings of the National Computer Security Conference*, Vol 14, 147-156, October, 1991.

[11] Dallas Semiconductor, "Java-powered Ring," http://www.ibutton.com/store/index.html#jring.

[12] Various, "Linux Security Module Project," http://lsm.immunix.org, 2001.

Solaris Bare-Metal Recovery from a Specialized CD-ROM and Your Enterprise Backup Solution

Lee "Leonardo" Amatangelo – Collective Technologies
W. Curtis Preston – The Storage Group, Inc.

ABSTRACT

The bane of all system administrators is the crashing of a mission critical system. Further depression sets in when the crash has caused the boot disk to become corrupted and no longer possess the ability to boot the system. A crashed mission critical system requires immediate attention. Literally, every moment of downtime equates to lost revenue. The desire is to get the mission critical system back to its normal functioning state in the quickest amount of time. The rebuilding of a computer system that has lost its capability to boot is known as a bare-metal recovery. The system will need to be built starting from a bare disk drive up to a bootable and functioning operating system.

Not until the release of Solaris 8 did the Solaris operating system contain a utility for providing bare-metal recovery functionality. Such functionality can be found in the operating system of other UNIX variants, such as IBM AIX, HP HP-UX, and Linux. However, by making use of the following Solaris utilities: ufsdump, ufsrestore, dd, cpio, tar, format, prtvtoc, fmthard, installboot, and JumpStart, bare-metal recovery can be achieved. Furthermore, it can be achieved by using a custom-built bootable CD-ROM (or DVD-ROM) and your environment's networked enterprise backup solution, such as Legato NetWorker and Veritas NetBackup.

Introduction

There is an ever-increasing demand to have mission critical computer systems run 24 hours by 365 days a year. When a mission critical system suffers from corrupted disk data, and when the disk drive in question happens to be the boot drive, the situation becomes more involved. One cannot simply run the backup application to restore the system because a functional operating system no longer exists on the system. This type of disaster recovery is known as bare-metal recovery. (The bare-metal comes from the early days of computing when storage media was a metal core. Nowadays, storage media is commonly magnetic or optical.)

Under such dire circumstances, rebuilding a boot disk can take what seems like an inordinate amount of time. The operating system needs to be installed, standard packages (clusters) installed, customized packages installed, patches applied, kernel tuning, and finally any special or custom configurations need to be set. To perform all of these tasks could take up to a couple hours.

Even using Sun Microsystems' automated installation utility, JumpStart, can still take what might be considered too much time, because following the installation of the operating system and any customized packages, the JumpStart process still needs to apply desired patches. The application of a large set of patches can take up to a couple of hours. The process is not totally complete because following the installation of the patches, any variable data will still need to be restored to the system using the environment's backup solution.

Until Web Flash in Solaris 8, Sun Microsystems did not provide a complete disaster recovery tool as an integral part of the Solaris product. As opposed to the IBM AIX UNIX environment, which has mksysb, and the Hewlett-Packard HP-UX environment, which has Ignite-UX and make_recovery. While not providing a formal disaster recovery (cloning) tool, Sun Microsystems does provide some very useful utilities to aid in the process of disaster recovery. Specifically, these utilities are ufsdump, ufsrestore, dd, cpio, tar, format, prtvtoc, fmthard, installboot, and JumpStart. When combined, these utilities can produce a very powerful disaster recovery tool.

This paper will discuss the steps needed to perform a bare-metal recovery on a Solaris system. In so doing, this paper describes a tool that enables the system administrator to perform a timely rebuild of a crashed system to the state of its last successful backup by using a custom-built CD-ROM and the environment's networked backup solution. Currently, the tool discussed in this paper is built to handle Solaris systems within an environment that uses Legato NetWorker or Veritas NetBackup. This tool can be extended to handle other shrink-wrapped or homegrown backup products.

Background Information

This paper is a by-product from the paper, *Unleashing the Power of JumpStart: A New Technique*

for Disaster Recovery, Cloning, or Snapshotting a Solaris System, [1] presented at the 14th Annual LISA Conference, (LISA-2000). The LISA-2000 paper and associated presentation discussed in detail how to create the "Capture and Restore Tool," (CART), a tool for capturing the image of a Solaris system onto a set of CD-ROMs, with the first CD-ROM volume in the set being bootable. Using this set of CD-ROMs, a server could be readily rebuilt onto the same hardware or cloned onto like hardware consisting of disk drives of the same or larger size as the original source system. The CART was achieved by using the same technique Sun Microsystems uses for installing the Solaris operating system from its installation CD-ROM. The technique incorporates a specialized JumpStart mechanism built in the CD-ROM.

By using the CART technique, it is possible to build a bootable CD-ROM (or DVD-ROM) that will automatically invoke a customized script whose function is to perform a bare-metal recovery using the latest filesystem backups captured for that system by the environment's backup solution. The CART restored the system's data from a set of CD-ROMs created during the capture phase. This new tool performs the bare-metal recovery by restoring data over the network from the environment's backup solution. This new ancillary tool for performing the bare-metal recovery of a Solaris system is referred to as the Bare-metal Ancillary Recovery Tool, or BART.

The concept of the BART was first announced at LISA-2000. At that time, the tool was under development and showed promising results. Since then, several requests from System Administrators and Backup & Recovery professionals have rolled in regarding the availability of this tool. At this time, we are now prepared to fully disclose the details of the BART, how it was built, and how it can be customized to address the idiosyncrasies of a particular environment.

Basic Bare-metal Recovery on a Solaris System

The use of any disaster recovery or bare-metal recovery tool does not replace the need for a complete Disaster Recovery Plan (DRP). Devising a good disaster recovery plan is hard work. It needs to be built from the ground up and it can take years to perfect. Since computer environments change constantly, the DRP must continually be tested to ensure it still works in the changed environment.

W. Curtis Preston's O'Reilly book [4] is a great resource on backup, recovery, disaster recovery, and bare-metal recovery. He walks the reader through the what, when, why, how, how many, and how often data on a system should be captured prior to the need for restoring it.

Since the topic of bare-metal recovery has been covered by other sources, specifically [3] and [4] listed in the references section, this paper will not discuss the topic in great detail. Instead, a list of pertinent steps for accomplishing bare-metal recovery on a Solaris system is provided.

Prior to the Disaster

1. Save the Volume Table of Contents, vtoc, of all disk drives on the system by using the prtvtoc utility which is very effective for capturing and saving this information to a file.
2. Save the /etc/vfstab file.
3. Save all appropriate metadata database information (which maps physical devices to logical devices). This information proves valuable when physically reconfiguring the hardware of a system.
4. Capture good backups of all filesystems on all disk drives on the system by using utilities such as: ufsdump, dd, cpio, tar, or third party backup software packages.

After the Disaster

1. Replace all defective disk drives and all other defective system components.
2. Replace the vtoc on all replaced disk drives by using the fmthard utility and the file saved in step (1) in the list of actions to take "Prior to the Disaster."
3. Boot the system to be recovered either via CD-ROM or from a boot server to place a functioning operating into memory on the system.
4. Mount the disk that is to be the new boot disk.
5. Recover the operating system to the mounted disk by using the same utility used in step (4) in the list of actions to take "Prior to the Disaster."
6. Place the boot block on the mounted disk by using the installboot command; this is a very important step else the disk will not be bootable.
7. Reboot the system.
8. Pray.

The Bare-metal Ancillary Recovery Tool – The "BART"

The Bare-metal Ancillary Recovery Tool, BART, consists of a single bootable CD-ROM. Like the CART, the BART contains a trimmed down version of the Solaris Software Installation CD-ROM. To create the BART, selected files were copied from the Solaris Installation CD-ROM to a read-writable hard disk drive. Scripts were borrowed from the CART project to accomplish the task of copying files to the "image disk." This disk drive is known as the "image disk" because once the disk is put into its desired configuration, it becomes the image that gets written to a recordable CD (CD-R or CD-WR). Since, the maximum capacity of a CD-ROM is about 650 MB, a one GB disk drive is adequate for the "image disk." The full layout, sizes, and description of the slices for the BART "image disk" are displayed in Diagram 1.

Description of the BART "Image Disk" slices

Slice s0 contains a trimmed down version of the Solaris Installation CD-ROM slice s0 and is present solely to enable the JumpStart mechanism.

Slice s1 contains the mini-root along with an adequate set of UNIX utilities. Upon booting the BART CD-ROM, slice s1 gets placed into memory and contains the mini operating system so that the system can function even though there is not anything on the disk drive(s) yet. The customized portion of the boot process accomplished through the custom JumpStart BEGIN script will eventually load the backup software package(s) into slice s1. At that point, the system will be able to access functioning backup software needed to accomplish the restore of the entire boot disk using the environment's backup solution.

Slices s2-s5 contain the boot information (bootblock) for the various hardware architectures of Sun Microsystems' products that run Solaris. The following file is also contained in these slices:

```
.SUNW-boot-redirect
```

which simply contains a single byte, the character '1', to direct the firmware boot PROM program to look for the kernel on slice 1 of the boot device.

Slice s6 is for future enhancements and could contain environment specific configuration and profile files to minimize or eliminate user interaction with the BART during a bare-metal recovery.

Lastly, slice s7 contains compressed tar files of the backup software packages, such as Legato NetWorker and Veritas NetBackup. These files get uncompressed, untarred, and placed into the appropriate directories resident in memory by the customized JumpStart BEGIN script.

Location of the Pertinent JumpStart files on the "Image Disk"

The pertinent JumpStart files that allow for the Solaris Install-like boot process to take place are located on the "Solaris Installation CD-ROM," and now the BART "image disk," in the following location (Solaris 2.6 is used in this example):

```
/s0/Solaris_2.6/Tools/Boot/usr/
    sbin/install.d/install_config
```

The pertinent standard issue JumpStart files found in this location are the following:

rules.ok: JumpStart Installation "RULES" file

install_begin: JumpStart Installation "BEGIN" script; called out by the "rules.ok" file

devsyn_finish JumpStart Installation "FINISH" script; called out by the "rules.ok" file

The BART will replace the rules.ok and the install_begin files with its own versions. It is important to name the BEGIN script the same as the BEGIN script being called out in the customized version of the rules.ok file. Similar to the CART, the BART does not need to call out a FINISH script, and thus, the devsyn_finish is removed from the "image disk." Correspondingly, the rules.ok file does not make reference to a FINISH script. Since, the customized BEGIN script ends with a reboot of the system, even if a JumpStart profile or JumpStart FINISH script were placed in the rules.ok file, they would never get invoked.

Customized JumpStart BEGIN script actions

Upon booting from the BART CD-ROM, a mini-root operating system gets placed on the target system. The BART then proceeds through the customized JumpStart BEGIN script, "bart_begin" to perform the bare-metal recovery. Some of the more salient actions performed by this BEGIN script are outlined below:

1. The BART queries the user whether to run in interactive mode or profile mode. If interactive mode, the BART further queries the use for specifics regarding the environment:
   - Legato NetWorker or Veritas NetBackup
   - Name of master backup server
   - IP Address of backup server
   - Name of target system
   - Name target system is known as by the backup server
   - IP Address of target system
2. Creates a fully writable directory, /tmp/BART_custom
3. /etc/inetd.conf file needs to be installed
4. If appropriate, uncompresses and untars "networker.tar.gz"; place an installation of Legato NetWorker under /nsr
5. If appropriate, uncompresses and untars "openv.tar.gz"; place an installation of Veritas NetBackup under /usr/openv
6. If running NetWorker, need to start daemon through /etc/rc2.d startup scripts

Slice	Partition	Contents	Size
0	a	Install Directories & Dist.	160 MB
1	b	Mini-root	40 MB
2	c	Boot Info – sun4c	1 cylinder
3	d	Boot Info – sun4m	1 cylinder
4	e	Boot Info – sun4d	1 cylinder
5	f	Boot Info – sun4u	1 cylinder
6	g	Config + Profile Files	10 MB
7	h	Compressed tar files of Backup S/W Packages	440 MB

Diagram 1: The BART "Image Disk" layout.

7. Discover Network Interfaces
8. Display network interfaces found and ask which default network interface to use
9. Display a default IP subnet netmask and ask which default IP subnet netmask to use
10. Activate specified network interface
11. Ask which BACKUP Program to use:
    a. Legato NetWorker
    b. Veritas NetBackup
12. Ask for the master backup server hostname
13. Ask for the master backup server IP Address
14. Possibly use nslookup to gethostbyaddr then display IP Address or "Don't Know IP Address"
15. If NetBackup, then
    a. ask for client hostname which the backup server knows the client
    b. bp.conf file format:

```
SERVER = $master_name
     (received from query)
CLIENT_NAME = $client_name
     (received from query)
```

    c. Place the master_name, client_name, client_name_server_knows_as information into the /etc/hosts file
16. If NetWorker, then
    a. query for NetWorker server name and build /tmp/startup
17. Have we been running the save_root_vtoc.sh? If YES,
    a. Recover from backup vfstab → /tmp
    b. Recover file to /tmp
    c. Format disk drive with file

    If NO,
    a. Query for disk partitioning information
18. Partitions the disk(s) appropriately using the target system's disk partition table.
19. Creates filesystems on the partitions.
20. Utilizes the local Backup software to restore from the enterprise backup server data for all the filesystems on the disk(s).
21. Places appropriate bootblock on the boot disk drive.
22. Reboots the system.

Requirements of the BART

1. Saved copies of volume table of contents, vtoc, for all disks on the target system.
2. A saved copy of the target system's /etc/vfstab file.
3. A good backup of all data for the target system exists in the environment's backup solution.
4. Run the save_root_vtoc.sh on all hosts in the environment everyday and save the info.
5. A CD-ROM drive exists on the target system.
6. The target system can access the network.
7. The target system is known by the environment's backup server.

How the BART Functions in the Networked Environment

Diagram 2 depicts how the BART works in a networked environment. The steps to rebuild or clone a target system involve the following:

1. Place the BART CD-ROM in the CD-ROM drive of the target system indicated by step (1) in the diagram and boot the system via the CD-ROM.
2. On the console of the target system, the BART will query the user whether the restore should get the environment information via interactive mode or from reading a profile. If the user selects the profile mode for data entry, several options are available. The next query is for the user to enter the full path to where the profile can be found; typically this will be either the floppy drive (2a) or an exported filesystem from an NFS BART profile server (2b). A secondary CD-ROM device could also be used, but most systems do not possess more than one.
3. By gleaning information from the interactive mode or the profile mode, the target system will

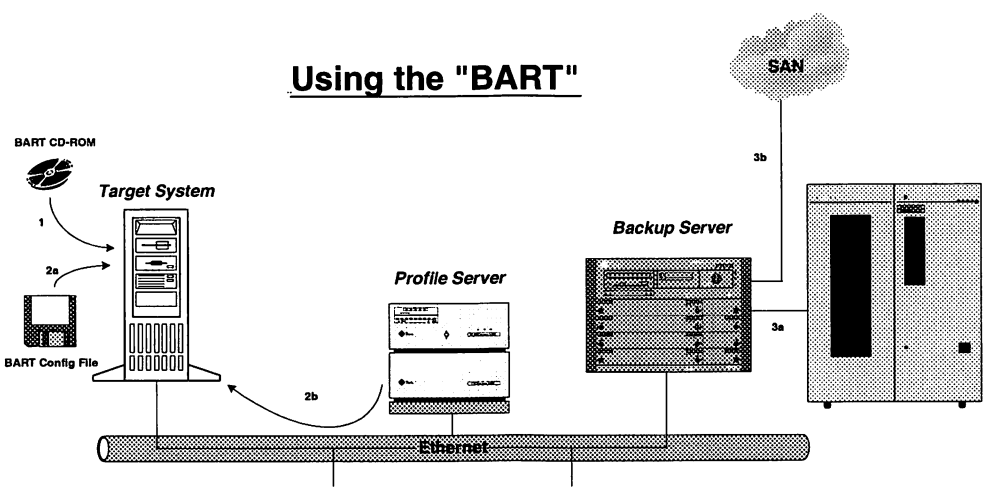


Diagram 2: How BART works in a networked environment.

be able to locate the appropriate backup server in the environment. At this point, BART does not care whether the data comes from direct attached storage (3a) to the backup server or from a Storage Area Network, SAN, (3b).

Testing the BART

The authors have used and customized the BART at various Fortune 500 clients, especially W. Curtis Preston, who has been involved in designing and implementing their Enterprise Backup & Recovery and Enterprise Disaster Recovery solutions. The BART has been a proven success at the clients where it has been employed.

Limitations of the BART

1. Currently implemented with Legato NetWorker and Veritas NetBackup. The tool can also be developed to handle other backup software products as well.
2. Currently, only handles Solaris operating system due to the special JumpStart feature.

Conclusion

For large enterprise sites with elaborate disaster recovery plans that include data mirrored to remote locations, the BART may not prove to be of value or of need. However, for the small to medium sized enterprise where budget constraints have not allowed for the desirable disaster recovery plan, or for the large site that does not have elaborate disaster recovery plans implemented, this tool may very well prove to be a life saver.

Similar to the CART, the BART can also be implemented on a networked JumpStart Server. However, the BART was specifically developed for the consultant who specializes in Backup and Recovery administration. By building the BART on a bootable CD-ROM, this professional can use it at any client site without having to setup or get familiar with an existent JumpStart server in the client's environment.

Resources

The following freeware products from Joerg Schilling [9] were used in the development and implementation of the BART:

cdrecord: A program for creating single/multiple session CD-R on a SunOS, Solaris, Linux, *BSD/SGI, HP-UX, AIX, NeXT-Step, or Apple-Rhapsody system.

sformat: A program to format/analyze/repair SCSI hard disks on a SunOS, Solaris, or Linux system.

scg: A driver to send any SCSI command to any SCSI device on a SunOS or Solaris system.

fbk: A driver to mount a file containing a filesystem; (File simulates Block device on Solaris).

mkisofs: Puts files in ISO-9660 format.

A "Smart and Friendly" CD-RW 426 Deluxe CD-Recorder was used in the development and final implementation. There were not any issues encountered with the installation or use of the cdrecord products or with the use of the "Smart and Friendly" CD-RW 426 Deluxe CD-Recorder. Both of these products receive a high endorsement from the authors.

Other CD-R recording hardware and software products (i.e., Young Minds, Inc., HyCD, Gear to name a few) could have been integrated into the CART as well. However, the price of cdrecord and its associated products could not be beat.

Also, of great value to the development of the BART is the following freeware product from Matthew R. Green:

mksunbootcd: combines filesystems for Sun Microsystems computers for creating bootable compact disc images.

Acknowledgements

We thank The Storage Group, Inc. for proposing the initial concept of the BART and for providing the equipment necessary to design, build, and test it.

We thank the "Publications Group" of Collective Technologies for providing editing expertise for this paper.

We thank Adelaida Esquivel for also providing editing expertise for this paper.

We thank Joerg Schilling whose freeware products were indispensable in the creation and final product of the BART due to their ease of installation, use, and an unbeatable cost.

Author Information

Lee "Leonardo" Amatangelo was graduated from the University of California, Irvine in 1983 with a B.S. in Molecular Biology and in 1985 with a B.A. in Anthropology. He has been working in the computer industry since 1981. Currently, he is a systems management consultant specializing in Solaris and disaster recovery for Collective Technologies. He can be reached via email at leonardo@colltech.com or lamat@earthlink.net and by physical mail at Collective Technologies, 9433 Bee Caves Road, Building III, Austin, TX 78733.

W. Curtis Preston is the President of The Storage Group, Inc. (http://www.thestoragegroup.com), and has been specializing in storage for over seven years and has designed, implemented, and audited enterprise-wide backup and recovery systems for many Fortune 500 and e-commerce companies. His O'Reilly & Associates book, UNIX Backup & Recovery, has sold over 20,000 copies, and he writes a regular column for UnixReview online and SysAdmin magazine. Curtis is also the webmaster for backupcentral.com, and can be reached at curtis@thestoragegroup.com.

References

[1] Amatangelo, Lee "Leonardo," "Unleashing the Power of JumpStart: A New Technique for Disaster Recovery, Cloning, or Snapshotting a Solaris System," *LISA XIV Conference Proceedings*, 2000.

[2] Kasper, P. A. and A. I. McClellan, *Automating Solaris Installations – Custom JumpStart Guide,* SunSoft, Prentice Hall, 1995.

[3] Nemeth, E., G. Snyder, S. Seebass, and T. Hein, *UNIX System Administration Handbook*, Second Edition, Chapter 9, Prentice Hall, 1995.

[4] Preston, W. Curtis, *Unix Backup & Recovery*, O'Reilly and Associates, Inc., 1999.

[5] Sun Microsystems, *Solaris 2.6 – Solaris Advanced Installation Guide*, Revision A, Mountain View, CA, Part No. 802-5740-10, August 1997.

[6] Sun Microsystems, *Solaris 8 – Advanced Installation Guide,* Mountain View, CA, Part No. 806-0957-10, February, 2000.

[7] Zuberi, A., "JumpStart in a Nutshell," *Inside Solaris*, Chapter 1, February, 1999,

[8] http://www.fadden.com/cdrfaq/faq00.html#[0-1].

[9] http://www.fokus.gmd.de/research/cc/glone/employees/joerg_schilling/private/.

[10] http://www.smartandfriendly.com/.

[11] http://www.ymi.com/.

Accessing Files on Unmounted File Systems

Willem A. (Vlakkies) Schreüder – University of Colorado, Boulder

ABSTRACT

This paper describes a utility named `ruf` that reads files from an unmounted file system. The files are accessed by reading disk structures directly so the program is peculiar to the specific file system employed. The current implementation supports the *BSD FFS, SunOS/Solaris UFS, HP-UX HFS, and Linux ext2fs file systems. All these file systems derive from the original FFS, but have peculiar differences in their specific implementations.

The utility can read files from a damaged file system. Since the utility attempts to read only those structures it requires, damaged areas of the disk can be avoided. Files can be accessed by their inode number alone, bypassing damage to structures above it in the directory hierarchy.

The functions of the utility is available in a library named `libruf`. The utility and library is available under the BSD license.

Introduction

There are many important reasons for being able to access unmounted file systems, the prime example being a damaged disk. This paper describes a utility that can be used to read a disk file without mounting the file system. The utility behaves similar to the regular `cat` utility, and was originally named `dog`, but was renamed to `ruf` for **r**eading **u**nmounted **f**ilesystems to avoid a name conflict with an older utility.

In order to access an unmounted file system, the utility must read the disk structures directly and perform all the tasks normally performed by the operating system; this requires a detailed understanding of how the file system is implemented. Implementing this utility for a particular file system is an interesting academic exercise and a good way to learn about the file system. The original work on this utility was in fact done in Evi Nemeth's system administration class.

Boot Block	Cylinder Group 1	Cylinder Group 2	...	Cylinder Group *m*
1	*k blocks*	*k blocks*		*k blocks*

Figure 1: File system layout.

Most Unix systems – including HP-UX, Solaris, Linux and the BSD family – use a general purpose file system derived from the Fast File System (FFS). [McKusick, et al., 1984] While the Linux ext2fs implementation differs considerably from the FFS, the concepts it uses are basically the same.

Figure 1 shows the layout of the file system. The file system, which may be all or part a physical disk, consists of a boot block followed by a number of equal sized entities called cylinder groups. The physical data blocks that make up a cylinder group are organized such that they can be read with minimal movement of the disk heads, thus improving performance. In fact, the FFS is itself derived from the original System V File System (S5FS); the major difference between the FFS and S5FS is that the S5FS has a single cylinder group. Cylinder groups are relatively small, often tens of megabytes in size. Contemporary file systems may therefore contain hundreds or thousands of cylinder groups.

All disk drives organize data into fixed sized physical blocks. The file system uses blocks that are one or more physical disk blocks. The terminology of the FFS is a bit confusing, in that a block consists of several fragments or frags. A physical disk block may therefore really correspond to a fragment and not to a block. For example, a 4 kB block may consists of four 1 kB fragments. The block address actually refers to the fragment address, so that in this example with four fragments per block, the addresses of the blocks are 0, 4, 8, etc., so that each address readily decodes to the correct block or fragment.

Super Block	Accounting Information	Inode Table	Data Blocks
1	*k blocks*	*l blocks*	*n blocks*

Figure 2: Cylinder group layout.

The layout of a cylinder group is shown in Figure 2. The superblock contains many critical file system parameters such as the block size, number of fragments per block, and other information needed to access the file system. Since these parameters are critical to the integrity of the data on the disk, several backup copies of the superblock are spread over the disk, potentially one per cylinder group.

Most information about a file is saved in a structure called an inode. This information includes the file owner, permissions, times, size and so on. Inodes are of fixed size and are numbered sequentially. Given

the inode number, a somewhat complicated algorithm gives the exact location of the inode on the disk.

The file contents are stored in data blocks. A list of the data blocks associated with a file is stored in the inode. Reading a file involves reading the inode, extracting the list of data blocks, then reading the actual data in the blocks.

The typical tree structured directory system is implemented as a sequence of files. A directory is nothing more than a special file that associates file names with inode numbers. To read a file by its file name, the root file that appears at a fixed inode number is read. Sub-directories appear as records in the file with associated inodes. Marching the directory tree requires reading sub-directories until the specific file and the associated inode number are found.

Program Organization

The `ruf` program is organized into three sets of functions. At the lowest level, there are the device functions: These functions read bytes from the disk at boundaries and in sizes appropriate for the device being accessed. They also buffer data to improve performance. While important to the operation of the program, this paper simply asserts the ability to read arbitrary byte offsets on the disk using a 64 bit address.

The next set of functions are the file system functions, where the majority of work is performed. At this level, peculiarities such as disk, inode and directory structures are resolved. The functions `fsmount` and `fsumount` read the superblock and perform memory management functions, analogous to the `mount` and `umount` system calls. Alternate superblocks may be read with appropriate command-line parameters. The functions `fsopen`, `fsread`, and `fsclose` are used to open, read and close a file based on an inode number passed to `fsopen`. The function `fsscandir` is implemented to read directory files using `fsopen`, `fsread` and `fsclose`.

The last set of functions take care of the walking the directory structure, resolving symbolic links and listing the file or directory. The function `walkpath` takes a path and finds the corresponding inode number by reading each of the directories in the path. Symbolic links causes `walkpath` to be called recursively. The function `stream` lists the contents of a regular file to standard out, or performs a directory listing looking like `ls -il` to standard out. Finally, a function `ruf` provides a convenient interface to all the lower level functions.

The functions are implemented in a library named `libruf`. A simple main program calls functions in `libruf`, but the functionality may be readily incorporated into other programs.

Porting `libruf` to a new file system derived from the FFS should only require specifying which header files to include and definition of a few macros in the files fs.h and fs.c.

The Superblock

The first step in accessing the file system is reading the superblock. This is done through the function `fsmount`, which returns a private structure that is used in subsequent operations.

The primary superblock is stored near the beginning of the file system. At the very beginning of the file system is typically 1 kB of boot information. The superblock is at a fixed offset from the beginning of the file system, typically 1 kB. When using 1 kB blocks, the boot information is contained in the first block, and the superblock appears in the second block. Larger blocks contain both the boot information and superblock. Reading the primary superblock simply requires seeking to the superblock offset and reading the required number of bytes.

Reading an alternate superblock is somewhat of a chicken and egg problem. The alternate superblocks appear at the beginning of some subsequent cylinder groups. However, in order to determine the cylinder group size, the information in the superblock is required. Fortunately, cylinder groups are not of arbitrary size. One of the important components of the cylinder group is a set of bits indicating used blocks. This set is stored as a single block. Since a 1 kB block contains 8192 bits, the cylinder group for this block size will often contain 8192 blocks.

In order to read an alternate superblock, it may be necessary to try a few of the common block sizes in order to find the superblock. The function `sbfind` reads all the blocks on the disk and prints fragment addresses where the superblock magic appears at the correct offset. If the root superblock is readable, the function `sblist` can be used to read it and then test all the appropriate locations and print the fragment addresses where superblocks occur.

In older implementations, every cylinder group contained a copy of the superblock. With the explosive growth in disk sizes, the number of cylinder groups have grown so large that newer implementations store only a dozen or so copies of the superblock.

Much of the superblock information is intended to specify parameters that optimize disk performance and facilitating writing to the file system. The number of parameters required to read the file system is actually a very small subset of the values in the superblock. The number of bytes per block, number of bytes per fragment and number of fragments per block are critical parameters. Furthermore, the number of inodes per cylinder group is required to determine which cylinder group contains the inode. Finally the number of blocks per cylinder group determines the cylinder group boundaries. These parameters are saved as part of the private structure returned by `fsmount`. This structure must be passed to all functions that read the file system. When done, the program should call `fsumount`. This function frees

allocated memory and closes the device file used to read the file system.

Locating the Inode

The most difficult part of reading a file is locating the appropriate inode. File systems directly derived from the original FFS define convenient macros to aid in this process. Others such as the ext2fs do not.

All cylinder groups contain the same number of inodes as defined in the superblock. Call this *ipg* for inodes per group. The cylinder group of an inode for the ext2fs is then $(inode - 1)/ipg$, while for the others it is *inode/ipg*. The offset of the inode index within the cylinder group is $(inode - 1)\%ipg$ for the ext2fs, and *inode%ipg* for the others.

The inodes are stored sequentially in the cylinder group, as shown in Figure 2, in the block labeled "inode table." In file systems other than the ext2fs, the inode table starts at an offset defined by the cgimin macro. This offset is actually stored in the superblock.

For the ext2fs, there is no convenient macro or superblock parameter. Figure 3 illustrates the layout of an ext2fs cylinder group in detail. In order to determine the offset of the inode table, the number of blocks consumed by the other structures must be calculated. The superblock is offset *sboff* bytes from the start of each cylinder group, so if the offset superblock does not fit into a single block, it consumes more than one block. For a block size of *bs*, the superblock structure *sb* will therefore consume:
$(sboff + sizeof(sb) - 1)/bs + 1$ blocks.

A set of group descriptors, one for each cylinder group, occupies the next *k* data blocks. The number of cylinder groups is calculated from the total number of inodes on the disk *inod_count* as *inod_count/ipg*. The number of blocks *k* is therefore given by:

$$\frac{((inod\_count/ipg - 1) * sizeof(ext2\_group\_desc))}{bs} + 1$$

where *ext2_group_desc* is the group descriptor structure.

The next two data blocks are bitmaps for the data blocks and inodes, respectively. Therefore, the block offset at which the inode table is

$$\frac{((inod\_count/ipg - 1) * sizeof(ext2\_group\_desc))}{bs} + \frac{(sboff + sizeof(sb) - 1)}{bs} + 4.$$

Once the offset of the inode table is found, the offset of the inode of interest is simply the size of an inode times the number of intervening inodes. The inode is read by seeking to the appropriate location and reading the appropriate number of bytes, usually 128.

Reading the file

The inode contains all information about the file except the file name and actual file data. This includes the file attributes such as user and group identifiers, permissions, times, file type and size. The `fsopen` function locates and reads the inode and allocates a private data structure for the file. This structure is passed to subsequent calls which read the file. When done with the file the `fsclose` function is used to deallocate the structure.

Sequentially reading the file with `fsread` requires reading the data blocks of the file in order. The inode contains a list of (typically) the first twelve data blocks known as direct blocks. For small files, all data blocks can be addressed in this way.

There are typically three slots for indirect blocks. These slots are used for single, double and triple indirection. The single indirect block is a data block containing a list of pointers to data blocks. The double indirect block points to a data block containing pointers to indirect blocks. Each of these indirect blocks in turn points to data blocks. For even bigger files, triple indirection is used. Table 1 shows the maximum file size that can be addressed using different block sizes.

Block Size	Direct	Single Indirect	Double Indirect	Triple Indirect
512	6 kB	70 kB	8 MB	1 GB
1024	12 kB	268 KB	64 MB	16 GB
2048	24 kB	1 MB	513 MB	256 GB
4096	48 kB	4 MB	4 GB	4 TB
8192	96 kB	16 MB	32 GB	64 TB

Table 1: Maximum file sizes by block size.

The purpose of using both fragments and blocks is to optimize disk usage. For example, on HFS systems, the block size is typically 8 kB, consisting of eight 1 kB fragments. The block addresses are therefore multiples of 8. Hence, for large files consisting of mostly blocks, eight times fewer addresses are required than if fragments were addressed, thus decreasing the indirect block overhead. The very last block of the data typically does not fill a data block. Therefore the last data block address typically addresses only a fragment or sequence of fragments. The fragments are contiguous, so it is simply

Super Block	Group Descriptors	Data Block Bitmap	inode Bitmap	inode Table	Data Blocks
1+ blocks	*k blocks*	*1 block*	*1 block*	*n blocks*	*m blocks*

Figure 3: ext2fs cylinder group layout.

necessary to read the number of bytes remaining in the file from that fragment address in order to finish reading the file.

Data block addresses are absolute, so that a file may use blocks from more than one cylinder group if necessary. A block address of zero has special meaning. When this address appears in a direct or indirect block, it implies a block with all zeroes. This mechanism conveniently handles files with holes. In the read functions, this address is conveniently handled by a simple test that returns a data block with all zeroes instead of performing an actual disk read.

In current implementations, the ext2fs has only one fragment per block. Therefore, although it does store fragment parameters in the superblock, fragments and blocks are in fact the same.

Finding the File

A directory is simply a file with a defined structure. The original S5FS allowed only file names of 14 characters, so that each directory entry could be stored as a fixed-length record; file systems with this limitation are rare today. A typical directory is made up of a sequence of variable-length records; each record consists of an inode number, record length, name length and the name itself. Some file systems such as FFS and later versions of the ext2fs also store the file type in the directory record, duplicating the information stored in the inode. A record with an inode of zero is a placeholder and is skipped. The record length will often be greater than necessary, as deleted directory entries are subsumed by extending the previous directory entry.

The function `fsscandir` is used to read the directory. The directory is identified by its inode number; when a directory entry to find is specified, the inode number of that entry is returned. If no entry is specified, a directory listing is produced instead.

File names, as stored in directories, do not contain path information; instead, only the base file name is stored. For example, in the /etc directory the file /etc/passwd will simply be named passwd.

All files are accessed through the root directory, which is always at a known inode number. The function `walkpath` parses the path into its components, and reads each directory in turn with `fsscandir` to find the inode of the next entry in the path. This entry is associated with an inode number, denoting the subdirectory, symbolic link or the final file to be read.

When a file system is not mounted, the path names are relative to the root of the file system. For example, when the /usr tree is in a separate file system, the file /usr/bin/gcc will appear as /bin/gcc in this file system.

Links and Devices

On UNIX systems, everything is a file. Devices are represented as files with a special file type. The kernel addresses these files using a major and minor number stored in the inode, and there are no associated data blocks. The `ruf` program simply reports these special files as device files.

Hard links exist when more than one entry to an inode exists in the directory files. Hard links require no special treatment when reading a file. Symbolic links, also known as soft links, however, are special files. Instead of containing actual file data, the symbolic link references another file or directory. A symbolic link may point to a file not on the disk in question. When reading a directory tree, encountering a symbolic link requires that the current leaf in the file path be replaced by a new, arbitrary path. This is achieved by recursively calling `walkpath`. The explicit occurrence of the .. file name in each directory allows resolution of the inodes in linear progression.

Symbolic links are a significant overhead in resolving path names; to improve performance, short symbolic links can be stored in the inode itself, where the data blocks would be stored (since no data blocks are needed). Typical file system implementations allow symbolic links of up to 60 bytes to be stored this way; symbolic links that exceed this length is stored in a data block. Strangely, while both UFS and HFS suggests that this can be done, all symbolic link names are stored in a data block, regardless of the length of the symbolic link.

Usage

The `ruf` utility behaves much like the regular `cat` utility, except that it takes the device name as the first argument before the file names. For example, on an HP-UX system with the root system on c0t6d0, the file /etc/fstab can be read using either

```
ruf /dev/dsk/c0t6d0 /etc/fstab
```

or

```
ruf /dev/rdsk/c0t6d0 /etc/fstab
```

Note that both the character and block devices can be used. On a Linux system with /dev/hda8 containing /var, the file /var/log/messages is read using

```
ruf /dev/hda8 /log/messages
```

Alternately a directory listing of the /var/log directory can be obtained using

```
ruf /dev/hda8 /log
```

Assuming this listing shows messages to have inode number 13294, the file can be read using

```
ruf /dev/hda8 13294
```

An integer file name is assumed to be an inode number.

This particular file systems uses 4 kB blocks. The same file can be read using the alternate superblock at fragment 98304 using

```
ruf -s98304:4096 /dev/hda8 /log/messages
```

or

```
ruf -s98304:4096 /dev/hda8 13294
```

The values of important file system parameters such as the block and frag size, blocks, fragments and inodes per group and number of cylinder groups can be read by omitting the file name, for example

```
ruf /dev/hda8
```

When the root superblock is readable, the fragments of all the alternate superblock locations can be listed using

```
ruf -l /dev/hda8
```

If the root superblock is not readable, the entire disk can be searched for potential superblocks using

```
ruf -f4096 /dev/hda8
```

Conclusions

The `ruf` utility is useful for accessing unmounted file systems. It is also very instructive in learning implementation details of various file systems. The functionality can be embedded in other programs through the `libruf` library. All the software is available under the BSD license at http://www.netperls.com/ruf.

About the Author

Vlakkies Schreüder holds a Ph.D in computational fluid dynamics and is currently a senior engineer with Principia Mathematica where he works on solving practical problems in fluid flows. He is also working on a Ph.D in parallel systems at the University of Colorado, Boulder. Reach him at vlakkies@colorado.edu.

Acknowledgments

I want to thank Evi Nemeth for putting me up to this in the first place, and Adam Moskowitz for suggesting many improvements to the paper.

References

Bach, M. J., *The Design of the Unix Operating System,* Prentice-Hall Software Series, Englewood Cliffs, NJ, 1986.

Bovet, D. P., and M. Cesati, *Understanding the Linux Kernel,* O'Reilly, Sebastopol, CA, 2001,

McKusick, M., W. Joy, S. Leffler, and R. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, Vol. 2, Num. 3, pp. 181-197, August, 1984.

Automating Infrastructure Composition for Internet Services

Todd Poynor – Hewlett Packard Laboratories

ABSTRACT

This paper describes a framework for automatically configuring relationships that tie together the software and hardware infrastructure components for an Internet-based service, which we also call "composing" the service infrastructure. Management and infrastructure software employ the framework to specify, discover, and react to changes in the infrastructure components participating in the service, automatically configuring cross-component relationships based on this information. This plays a key role in increasing the flexibility of a highly dynamic service infrastructure by reducing manual configuration required to redeploy resources. The framework also facilitates enterprises distributed across multiple autonomous administrative domains by automating chores required to tie resources distributed among the domains together into a cohesive service. We advocate extending existing service discovery protocols to distribute the information needed by the framework and suggest this as an area for future standardization.

Introduction

The technology described in this paper arises out of our recent research into automation of deployment and configuration of Internet applications and enterprise infrastructure. Our chief motivation is to assist in manageability and flexibility of the data center. Especially for the more dynamic enterprise infrastructures envisioned for the near future [13, 14, 21], the configuration of the data center hardware and software may be in nearly constant flux to support changing workloads, changing customer sets, and changing hardware resources. Our goal is to redeploy available resources quickly and easily according to demand instead of massively over-provisioning dedicated resources, as has often been the usual practice. Today we find ourselves in the midst of a global slowdown in IT spending, and we can predict that a good many service providers will be interested in "doing more with less."

We are also interested in increasing reliability of management and consequent availability of the enterprise. Today, administrators must often manually perform a number of individual hardware and software configuration steps to effect a consistent change in data center configuration, which is a known source of errors and downtime. For example, a recent report [9] claims 40 percent of Web site downtime is due to "operations – e.g., not performing a required task or performing one incorrectly." Our goal is to pursue the extent to which the data center may instead be automatically reconfigured into the desired consistent state, in keeping with the Internet architectural principle of avoiding manually specified parameters [15].

Our work also targets management of a distributed enterprise hosted in distributed data centers, especially data centers operated under separate administrative domains by distinct service providers. For predictions of future trading of compute power as a "liquid commodity" [8, 10, 19, 22] to be realized, automated mechanisms for discovering and configuring the distributed components together into a cohesive enterprise are needed.

Among our first steps to address these topics is to design a framework for discovery and configuration of relationships among the infrastructure components, such as applications and networking functions, participating in an Internet-based service. Because this framework encompasses the problem of service advertisement and discovery, we suggest that this framework should leverage existing service discovery protocols.

The rest of the document is organized as follows. The next section describes our general approach. The following section discusses leveraging existing protocols for infrastructure discovery and describes a design leveraging the Service Location Protocol [1]. Subsequent sections describe our prototype, related work, and conclusions.

Automated Infrastructure Composition

This section describes our infrastructure composition framework in general, leaving aside for the moment details of any particular technology on which the framework may be based. We first define the concepts modeled by our framework and then describe the framework itself.

Describing an Infrastructure

The model chosen is of an enterprise described at a high level of abstraction by the following:

Resources, generally hardware, such as general-purpose server machines and load balancing appliances. A resource is identified by its network name (TCP/IP hostname).

Services provided by resources. We use the term *infrastructure services* to distinguish these from

the higher-level e-services that are built from these components. An infrastructure service may be identified both by a *service type* identifier that describes the standard service protocol and by a name that identifies the particular implementation, used to discover proprietary or version-specific features. For example, the server named server127.xSP.com (a resource) runs an HTTP server (an infrastructure service type) that is Apache 1.3.19 (a specific implementation). Other service *attributes* that aid in configuring communication with the service, such as to identify a non-default TCP port, may be specified along with the service identifiers.

Contexts in which the various infrastructure services act in concert to provide higher-level Internet services; services are tied together through acting in a common context. A context may be identified using an identifier, such as the name of a particular customer in a shared data center, that must match for disparate infrastructure services to participate in the same context. A context may instead be identified by a URL that points to more detailed configuration information about the context using such formats as XML. For instance, the HTTP server on server127.xSP.com serves Web site tom.com (a context), and participates in the set of infrastructure services, also including load balancers and a Web cache array, that cooperate to provide the tom.com e-service.

The context identifiers allow infrastructure services to select the proper remote infrastructure services to tie together into a higher-level service. Through being configured into common contexts, infrastructure services can discover the other participants in their contexts (and ignore those of other contexts). The various resources of a data center may act in different contexts, such as to be partitioned among different applications or to be employed on behalf of different customers, and resources may act in multiple contexts at a time, such as to be shared among multiple customers.

An extension of this model is that of a "virtual enterprise" comprised of multiple data centers, each of which may be operated by distinct service providers, as may occur if in the future compute power is traded as a commodity [8, 10, 19, 22]. The virtual enterprise need not manage, nor even be aware of, every resource and infrastructure service in every participating data center, but only the subset of these with cross-data-center interactions. (It is assumed each service provider manages the local infrastructure according to the specifications of the virtual enterprise customer, in order to scale to very large enterprises.) We therefore use one "global" context for cross-data-center relationships managed by the virtual enterprise and one "local" context for each participating data center that manages local resources and services for the virtual enterprise according to local policies. Each infrastructure service with cross-data-center interactions acts in both the global context for the virtual enterprise and the appropriate local context for the {virtual enterprise, data center} pair. Each infrastructure service with purely local interactions acts solely in the local context. For example, the tom.com virtual enterprise owns a Web switch that is programmed to direct requests to the data centers that are currently contracted to host the tom.com e-service. The Web switch and the external contact points of the various data centers (Web servers or load balancers or Web caches and so forth) are configured in the global context for tom.com. A participating data center's infrastructure that supports tom.com (such as back-end Web servers hidden behind a Web cache) is configured in the local context for tom.com within that data center.

An Infrastructure Composition Framework

This paper describes a protocol and framework by which the deployment of resources, services, and contexts is specified by management tools and discovered by the affected components. There are two sets of deployment information to manage: the persistent instructions on how services are to be deployed, as specified by an administrator or system management software, and the (possible subset of these) deployments currently running. We call the former "designated deployments" and the latter "current deployments."

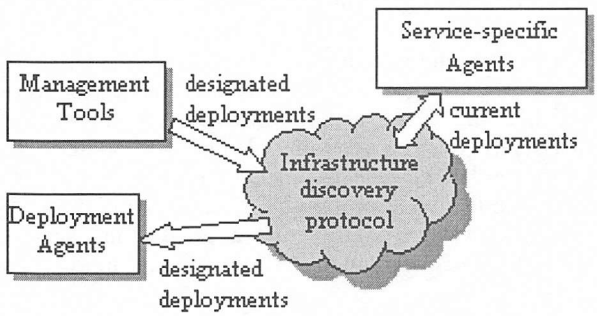


Figure 1: The infrastructure composition framework.

Management tools use the protocol to initiate changes in designated deployments. System management tools can provide high-level interfaces to model and modify the administrator's (or automatic management software's) designated deployments. The infrastructure composition framework automates the process by which the enterprise is configured to match those intentions. The eventual goal is that the administrator may execute a high-level command that reads something like "add a new Web server to the tom.com context"; the management tools and the composition framework then execute a number of configuration steps required to make that happen. Each resource runs one deployment agent that uses the designated deployment information to discover that resource's "personality," that is, what infrastructure services it is to provide and in which contexts. The deployment

agent initially queries this designated deployment information at boot time and then subscribes to notifications of updates. The deployment agent contacts the appropriate service- specific agents on that resource to start, stop, and reconfigure contexts for the services as instructed by changes in designated deployments.

Each service-specific agent thus executed then takes the actions needed to start, stop, or reconfigure contexts for the associated service. These changes in current deployments are then registered using the protocol, such that other service-specific agents can configure or deconfigure interactions with the redeployed service.

A newly started service-specific agent uses the protocol to discover other current service deployments and configures any interactions required based on that information. The agent also subscribes to notifications of changes in current deployments for the relevant service types, such that interactions can be reconfigured on an ongoing basis.

Figure 2 depicts a general-purpose server that runs two infrastructure services that participate in the protocol.

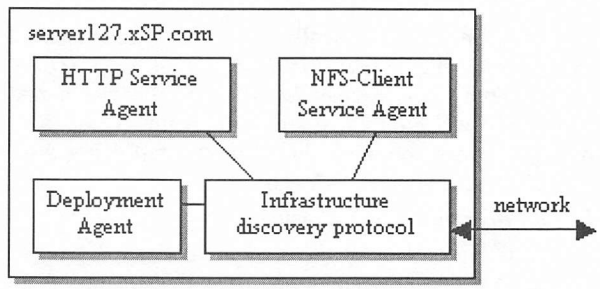


Figure 2: A server resource running two services.

The deployment agent discovers that it is to start the two services at boot time using the protocol and starts the services; the two service agents advertise their presence to other services and also discover and configure interactions with remote services using the protocol.

Figure 3 depicts two service-specific agents, one an HTTP load balancer and the other a firewall, reacting to notification of a new HTTP server deployed on resource server127 in context "tom.com" (in which context all participate).

The HTTP load balancer was originally told only to operate in the "tom.com" context, whereupon it discovered the appropriate set of target Web servers to balance among. Any change in the tom.com Web server membership is communicated to the load balancer agent so that it may update its set of targets. If an array of load balancers is employed then each instance may discover the other instances using our protocol, affecting the algorithms used to divvy up content; this is an example of discovering clustered instances of a specific product and configuring proprietary interactions.

A {resource, service type, context} 3-tuple defines the granularity at which deployments may be specified and discovered using the protocol, together with more detailed information made available through attributes associated with the tuple. The level of abstraction represented in the protocol is quite high-level, identifying the infrastructure services and resources participating in an Internet service together with basic information on how to establish communications. Further service-specific configuration information and actions may be addressed by more specific protocols or as extensions to this protocol through the service attributes or context information.

Further Details on the Framework

The deployment and service-specific agents for an appliance that do not support the infrastructure composition protocol natively may instead run on a nearby general-purpose server. The service-specific agent converts between the protocol and the proprietary management interface for the device. For specific-purpose appliances, the services to provide may be implicitly defined by the type of device and therefore the device need not query the designated set of services to execute. If so, at boot time a query is still generated for the resource's designated deployment, but only the information on designated context identifiers, not the service types, is used from the query result.

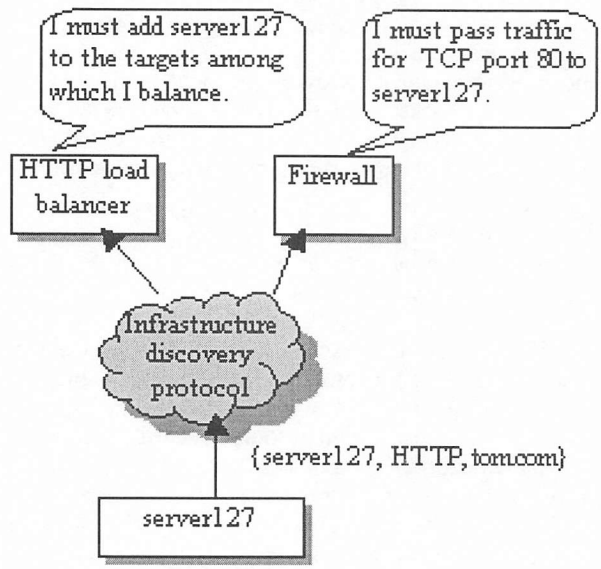


Figure 3: Two services configuring new relationships.

For general-purpose servers, the deployment agent and the service-specific agents may extend the existing means of starting and stopping services at system startup and shutdown time, such as the UNIX "init.d" scripts or the Windows "Services" applet. These are manually configured methods of specifying whether a certain service is to be started at boot time (or during normal system operation if manually

executed). As extended by our framework, services may instead be started or stopped at any time based on remote instructions. The existing service scripts or applets may also add to their existing actions (which include "startup" and "shutdown") the action of reconfiguring due to changes in the rest of the enterprise. We have thus far avoided any mention of specific communication protocols or encodings for the infrastructure composition protocol. This is because we wish to leverage existing technology for this, for which there are a number of choices, as discussed in the following section. The formats for specifying the names of resources, services, and context identifiers can vary depending on the underlying technology, such as strings within a URL or XML-based representations. Likely network transport protocols include UDP datagrams for local traffic and TCP for cross-data-center or other longer-haul traffic.

Leveraging Existing Discovery Protocols

Service discovery protocols serve the purpose of advertising information about available services, their attributes, and their access methods to prospective clients. Although our purposes are geared more toward enterprise management and automated discovery of relationships between component services, we share the concerns of representing, finding, and accessing services using standard names and attributes. Service discovery protocols are currently a fertile area of innovation and competition in the industry; the field does not need yet another protocol that encompasses this topic. Accordingly, our designs incorporate existing discovery protocols and APIs for the new purpose of infrastructure discovery, also resisting the temptation to invent a new API that is independent of the underlying discovery protocol.

Several existing protocols primarily target ad-hoc, non-centrally-managed networks, such as access to currently nearby services by mobile clients and networks of consumer electronics devices or office equipment. Among these are the Jini Lookup Service [3], the Simple Service Discovery Protocol (SSDP) [2], and Salutation [5]. Protocols oriented toward discovery of services in centrally managed networks include the DNS service location resource record [7] and the Service Location Protocol (SLP) [1]. Service advertisement and discovery protocols are also a part of the business-to-business frameworks E-Speak [4] and Universal Description, Discovery, and Integration (UDDI) [6]; many of our concerns regarding dynamic reconfiguration of intra-enterprise relationships can apply in a cross-enterprise context as well.

Our focus on enterprise management influences our designs in certain new directions. Services form relationships based on management policy (as expressed in designated deployments and context identifiers), rather than automatically choosing remote service peers from an available pool based on service-specific criteria (such as a print server that advertises the proper printer capabilities) or user selection from a menu of choices. Resources are explicitly identified in the protocol, rather than being implicitly defined in service location identifiers, and are represented as entities that may perform multiple services. Services are tied together by common context identifiers. Both current and designated service deployments are represented, such that the protocol may be used to determine the "personality" of the local resource. We also strive for stronger consistency guarantees than certain protocols that can deliver stale information and that do not enforce consistency across replicated repositories, since we should represent a consistent view of the enterprise for agreement among the various interacting components.

Leveraging the Service Location Protocol

This section presents an example design that leverages the Service Location Protocol (SLP) version 2 [1], a service discovery protocol on the IETF standards track, for infrastructure service discovery. SLP defines URL encodings for service deployments, called *Service URLs*. For example, an LPR protocol network printer named "queuename" on system "printserver" might be described by Service URL:

```
service:printer:lpr://printserver/queuename;\
        (media-size=na-letter)
```

This information is usually advertised and queried using UDP datagrams to *Directory Agents (DAs)*, which function as caches for service advertisements[1] Queries use LDAPv3 search predicates [25]. SLP messages may optionally be authenticated using digital signatures such as DSA [26].

Local Infrastructure Discovery

For infrastructure discovery, we keep the information on "designated deployments" separate from the information on "current deployments" using a new namespace (technically a "naming authority" string) for designated deployments named "conf". Deployment agents use the "conf" namespace to discover the services to be provided by the associated resource, then register deployed services and discover remotely deployed services using the usual namespaces.

Designated and current deployments are represented as Service URLs. Information on the associated resource and context for a service deployment is represented using attributes of the same names. Example Service URLs for designated and current deployments, respectively, of a service named tomapp on resource server127 as part of the tom.com context are:

```
service:tomapp.conf://futureuse;\
      (resource=server127.xSP.com),\
      (contexts=tom.com)

service:tomapp://server127.xSP.com;\
      (resource=server127.xSP.com),\
      (contexts=tom.com)
```

[1]We disregard the SLP small network model without Directory Agents for the enterprise-class purposes of this paper.

The "futureuse" identifier is a placeholder for the URL of further configuration information, the format of which we plan to specify in a future design phase. Resources are explicitly identified as an actual deployment attribute, rather than relying on the *addr-spec* portion of the Service URL, to facilitate searches on resource keys. Contexts are represented as attributes in part so that common mechanisms for queries on resource and context identifiers may be employed, although mapping contexts to the SLP feature of administrative "scopes" may suffice instead.

Both designated and current deployments are registered and unregistered using the usual SLP mechanisms, specifying Service URLs for the appropriate namespaces and that include the "resource" and "context" attributes.

To query for deployments, we use the SLP query by service-type name and attributes, extended to also allow queries for service-types that include the wildcard "*". For example, to discover the designated service and context deployments for a resource, a query is sent with service-type "*.conf" and a resource qualifier but no context qualifier. Queries for wildcard service-types also support arbitrary service types, the names of which are not well known but that register well known attributes, permitting configuration dependencies to be discovered. For example, a firewall can query for all services within the contexts in which the firewall operates, discovering the appropriate protocols and ports for non-well-known service types through the registered attributes.

Means by which software may be notified of changes in SLP service registrations are proposed in an IETF draft extension [12]. As above, we extend the design to allow subscribing to notifications of updates to arbitrary services, without knowing the appropriate service-type names in advance. Changes in designated deployments need be communicated to only the affected resources, which may need to start or stop services accordingly. We therefore extend the design such that subscriptions can be qualified by predicates such as "(resource=myhostname)". Once the services are actually started or stopped, the affected components are notified of the change in actual deployment as usual.

Distributed Infrastructure Discovery

SLP is designed to function within a single cooperatively managed network. We wish to adapt the protocol to represent the required service relationships across a "virtual enterprise" of distributed data centers. The two-level hierarchy of information to be managed, one level for the local data center (for which SLP is already designed) and one for the infrastructure services visible across the virtual enterprise, is expected to scale appropriately using SLP.

Preliminary designs call for at least one Virtual Enterprise Directory Agent (VEDA) that is a repository for information on all SLP Directory Agents (DAs) that participate at the global level. We call DAs that operate in the global context "global DAs" to distinguish these from DAs that operate solely in the local contexts of a data center, if the data center is arranged such that there are such "local" DAs. A VEDA is simply a global DA whose presence is guaranteed to exist for a sufficient period of time to "bootstrap" a new data center into the virtual enterprise – the distinction is necessary because we allow for the set of participating global DAs to be highly dynamic due to data centers joining and leaving the virtual enterprise as conditions warrant.

Initial discovery between a VEDA and the global DAs for a new outsourced data center joining the virtual enterprise is expected to occur out-of-band, perhaps as part of the process by which a contract is formed between the virtual enterprise and the data center. The new DAs and existing global DAs subsequently discover each other and exchange information on the relevant service deployments using SLP DA interaction protocols proposed in an IETF draft extension [16], further extended to encrypt communications to avoid giving too many clues to the "black hats." All cross-data-center SLP traffic occurs between global DAs; infrastructure services operating in a global context register information with a global DA that relays the information to remote global DAs. Figure 4 depicts Data Center Provider 2 joining a virtual enterprise. The new data center's global Directory Agent announces its presence to the Virtual Enterprise Directory Agent, which registers the new global DA and responds with a list of global DAs for the virtual enterprise. Only one peer DA is shown, hosted by the same provider as the VEDA. The new and existing DAs then exchange information on service deployments acting in the global context at the respective data centers and configure any cross-data- center interactions accordingly.

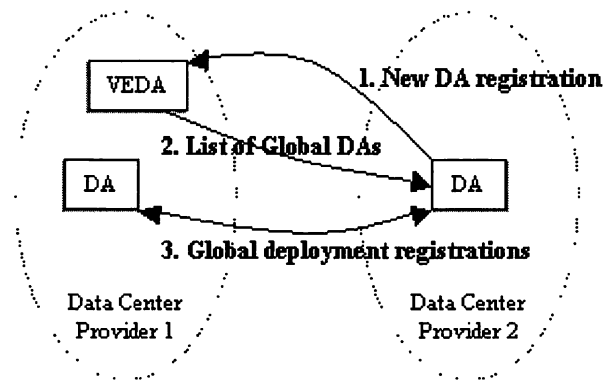


Figure 4: New data center joins the virtual enterprise.

Prototype Results

We prototyped a single-data-center subset of the framework, without support for cross-data-center interactions. The prototype infrastructure composition

protocol is based on OpenSLP 0.9.1 [23], an open source implementation of SLPv2, modified to support wildcard service queries as described previously. We also added a temporary measure for proposed deployment update notification extensions [11] not available at this time. The temporary mechanism simply executes the local deployment agent upon any service registration or deregistration received by a Directory Agent. Each resource therefore runs a DA (this is not normally the case in an SLP network); all updates are broadcast to each DA through DA interaction protocols built into OpenSLP. The deployment agent in turn informs each local service- specific agent of each update, which, of course, does not scale to realistically sized networks.

The resource and service components participating in the protocol are:

- An *ipfw/ipchains* firewall (and Network Address Translation in the future) on an intelligent network interface card in an HP J5000 PA-RISC server running HP-UX 11.0. All traffic to the following three systems is routed through this firewall.
- An NFS file server and boot server for the following two systems on an IA32 server running Linux 2.4.3.
- Two Apache 1.3.19 Web servers on two IA32 servers running Red Hat Linux 6.2. These systems boot over the network using Etherboot 4.6.3 and mount their file systems over NFS from the above boot/file server.

The above four server systems each run an SLP Directory Agent and have installed the infrastructure composition framework deployment agent and the appropriate service-specific agents. The deployment agent and the service-specific agents are implemented as bash/POSIX shell scripts to match the language of the existing service startup/shutdown scripts shipped with the operating systems (this is a somewhat clumsy language for this purpose; use of languages specifically targeted at configuration such as GNU cfengine [28] could prove beneficial). SLP registrations and queries are accomplished from a shell script by running a utility named slptool from the OpenSLP package that allows SLP protocol actions to be expressed as run string parameters. The service-specific agents leverage the existing system startup/shutdown scripts (such as */etc/rc.d/init.d/nfs* on Linux) to start and stop the associated services.

The server systems execute the local deployment agent at boot time to configure and start any services that participate in our framework. The deployment agent then queries the SLP ".conf" namespace to determine the designated services to start on the local resource, executing the service-specific script for each such service. The deployment agent is also called upon each change in designated or current deployments among the four systems, passing this information to the service-specific scripts.

In the absence of any fancier tools that abstract away the details of SLP syntax, we use the slptool utility to configure designated service deployments for any of the four systems. For example, an Apache Web server is started on system App4139 and configured for customer paladin.com using the following command on any of the systems:

```
slptool register http-server.conf:\
 apache//App4139 ''(resource=App4139),\
 (contexts=paladin.com)''
```

Upon receipt of the above registration, the SLP Directory Agent on App4139 runs our deployment agent, passing it the above information. The deployment agent in turn executes the http- server/apache service-specific script. That script configures an Apache "virtual host" for paladin.com (a mapping of the paladin.com domain name to a set of Web content and other Web site configuration information; there may be many virtual hosts served by the same Apache server) by modifying the httpd.conf file. The service-specific script also starts or restarts Apache using */usr/local/apache/bin/apachectl* and registers the new current Apache deployment using an slptool command similar to the above but without the ".conf" namespace.

The firewall service-specific agent is notified of the new http-server deployment on App4139, and opens access to TCP port 80 on App4139 via an *ipchains* command if not already. Conversely, if the http-server on that system is deregistered, access to the port is blocked.

The Web content for each customer is mounted over NFS from the file server. The NFS clients on the two Web server systems and the NFS server on the file server are services that participate in our framework. An NFS client operating in the paladin.com context locates the NFS server for paladin.com using an SLP query for the filesys-server:nfs service and mounts from the server hostname and file path as returned in the query results. The NFS server discovers the configured NFS clients through a query for filesys-client.conf:nfs registrations and allows access to the appropriate file system for the resources thus identified. Both NFS clients and servers are notified of updates in deployments and respond accordingly, and so a client will unmount a file system from a deregistered server and re-mount from a newly registered server; a server will drop access for a deregistered client and add access for a new client.

To move the paladin.com Web site from one Web server to the other, the old filesys-client:nfs and http-server:apache entries are deregistered (or modified to drop the paladin.com context) and new entries are registered for the new resource (or existing entries are modified to add the paladin.com context). The reconfigurations that ensue are as follows:

- The NFS server revokes access to paladin.com files for the old Web server system and grants access to the new system.

- The paladin.com files are unmounted from the old Web server system and are mounted on the new system.
- The Apache virtual host configuration for paladin.com on the old Web server is removed and a new virtual host is created on the new server.
- If there are no more contexts for the old Apache to serve then the Apache server is shut down and the firewall blocks port 80. If the new Apache server is not yet running it is started and the firewall opens port 80.

The above actions can be triggered by a command of form "move web site paladin.com from server1 to server2." The full set of actions has been observed to occur in less than two seconds. Add such components as network address translation, HTTP load balancing, Web caches, application servers and database servers, VLANs, and so forth to the picture and we can assert that the infrastructure composition framework results in a real benefit even for infrastructures that change only infrequently.

The prototype uses diskless servers such that in the future the protocol may also trigger software deployment activities. A notification to act in a new context causes the server to assemble and boot software needed to serve the associated context (such as the appropriate operating system and Web server software), in addition to mounting the appropriate files over NFS as currently prototyped. This is to be accomplished through a software deployment service that associates the context identifiers with NFS mount points for the OS, applications, and data, together with other configuration information required to deploy a new instance of a software stack previously configured for that context.

Another future direction is to represent dependencies between service reconfigurations on the local resource, such as to ensure each service using the file systems for a context have completed reconfiguration out of that context before the file systems are unmounted by the file system client. We are also looking at publishing additional service/context-specific configuration information over HTTP from URLs associated with the context identifiers. For example, when the service-specific agent for the Apache Web server is instructed to act in a new context it looks up such information as the administrator's e-mail address and authorization rules at URL http://context/config/http-server/apache.xml.

Related Work

We have previously discussed comparisons between our enterprise infrastructure composition protocol and a number of industry service discovery protocols, which function mainly for users to discover available end user services or for connecting office equipment or personal technology devices. In addition to these, the Secure Service Discovery Service (SSDS) of the UCB Ninja research project [8] targets users locating end user services over a wide area. Future directions for the Ninja project are to support purchasing services and compute power as commodities, at which time we can expect there will be an even stronger parallel to our work.

The IBM Océano research project [21] tackles a problem shared with our work: reconfiguring resources of a single data center that are dynamically allocated among multiple customers according to demand. For example, their paper mentions automatically reprogramming VLAN-based networks using SNMP when a server is reallocated among customers. The Muse project at Duke University [27] reprograms switches to serve requests from dynamic server sets allocated among customers. Their paper also discusses use of diskless servers for the same purposes of efficient reallocation across software environments as in our work. Further information on the protocols and mechanisms by which notification of state changes is performed and reconfigurations are triggered in these projects is not available to us at this time, but it seems a standards-based solution as we propose would fit perfectly.

Distributed infrastructure composition applies to various technologies termed "metacomputing", "grid computing", and "peer-to-peer". Although these technologies are primarily aimed at assembling large-scale computing power for supercomputing applications or at wide-area resource sharing or collaboration, the focus is beginning to also include enterprise applications and service providers. The emphasis is on such resource-oriented topics as discovery, description, allocation, and quality of service. Generally, applications employ new distributed resource access middleware or adaptations of existing distributed computing APIs such as MPI and CORBA. Darwin [18] concentrates on network resource management for distributed service providers; resources are tied together in a "virtual mesh" through brokers. Globus [19] and Legion [20] implement distributed computing middleware, including resource brokering and mechanisms for ongoing discovery of information on various aspects of resources in the distributed system, based on such protocols as LDAP and NIS. Our work differs from these in that it is focused on automated composition of service components in an enterprise infrastructure rather than on automated composition of resource components of a distributed application (over a partially pre-configured infrastructure). But it doesn't seem a far reach to adapt these technologies for our purposes. An automated infrastructure composition framework could complement these technologies for applications that interact with a dynamic set of peer applications or underlying infrastructure services. This could be achieved by extending the middleware associated with the technology or by leveraging emerging service discovery standards as we propose, with a possible advantage for lighter-weight standards-based

protocols that may lend themselves to use in a wider variety of devices.

Certain of these projects, and others such as the IETF Middlebox Communication Working Group [13], deal with protocols for communication of policy and/or quality of service information regarding a particular networking packet flow among the devices that participate in the conversation. For example, the appropriate firewall ports may be automatically opened for the duration of an IP telephony session. Automated configuration of end-to-end infrastructure for a particular service conversation is analogous to our goals of automated configuration of an enterprise infrastructure according to management policy, and may be a point of further future investigation. A number of projects, including FLASH [17] and previous work at NASA Ames [24], distribute service configuration files from a central repository, using facilities such as ONC NIS or UNIX r-commands (rcp, et al). The main goal is to keep multiple systems in sync rather than to dynamically form associations between systems, although such concerns as locating appropriate NFS servers do appear (the NASA Ames setup uses DNS resource records for this purpose, assuming the information does not change very often). These systems deal primarily with higher-level services configured using files and running on general- purpose servers that support file systems and file transfer protocols over an enterprise network that has been previously composed. The central site publishes detailed service-specific configuration information, rather than generalized information that is interpreted by service-specific agents on the remote nodes.

Conclusions

We have described a framework for automated discovery and configuration of the designated deployment of services on a resource based on centralized management instructions, and for automated configuration of interdependencies between infrastructure services. The management instructions describe designated service deployments in high-level terms ("deploy an Apache Web server on system server127 for customer tom.com"); the framework automates the process whereby the various enterprise components are reconfigured to match these instructions. Information on the membership of resources and services, as discovered through common context identifiers, is often sufficient to automate configuration of interactions between services, and can serve as a starting point for discovering more specific service configuration information via the published attributes and via descriptions associated with the published context identifiers.

We suggest that a standardized infrastructure subscribing to such a framework instead of proprietary administration interfaces would play a key role in more quickly reconfigurable, and more reliably reconfigurable, enterprise networks and services that scale to very large networks by distributing configuration intelligence among the various participant services. This technology can also serve as a foundation for virtual enterprise networks comprised of resources from a possibly dynamic set of disparate administrative domains by automatically configuring interactions needed across domains. We further propose extending existing statically configured mechanisms for starting and stopping services in general-purpose operating systems to handle such dynamism.

Existing service discovery protocols may be readily adapted for these purposes. It is our position that use of lightweight and standards-aligned protocols encourages adoption in the widest variety of devices and applications. In the case of the Service Location Protocol, proposed extensions for deployment notification subscriptions and for Directory Agent interactions, together with further recommendations in this paper for wildcard service queries and subscription predicates, would greatly enhance its usefulness for disseminating the information needed for automated infrastructure composition.

Acknowledgements

The author gratefully acknowledges Tom Wylegala, Lance Russell, Dejan Milojicic, Sven Graupner, Baila Ndiaye, and Dan Conway of HP Labs, and Jon Stearley and Paul Anderson of USENIX LISA, for their comments and suggestions on this paper. Thanks also to Lance for the iNIC and software interface and to Steve Hoyle and David A. Barrett of HP Labs for use of their diskless boot setup.

Availability

Bug fixes to OpenSLP developed during this effort have been contributed back to OpenSLP. OpenSLP enhancements and source code and documentation for the software used in our prototyping may be downloaded at http://www.hpl.hp.com/personal/Todd_Poynor/. There may also be a future effort to provide these features in Kempf & Associates SLP for Win32, see http://www.moeller-antik.com/~guttman/kaslp.htm.

Author Information

Todd Poynor has spent his entire professional career of 16 years at Hewlett-Packard in research and development of the HP 1000 Real Time Executive, the HP-UX kernel and system utilities, and most recently a number of technologies for computing platforms and Internet infrastructures. Reach him at postal address 1501 Page Mill Rd, Palo Alto, CA, 94304-1126 or at e-mail address todd_poynor@hp.com.

References

[1] Guttman, E., et al., "Service Location Protocol, Version 2," *IETF RFC 2608,* June 1999.

[2] Goldand, Yaron, et al., "Simple Service Discovery Protocol /1.0, IETF Internet-Draft draft-cai-ssdp-v1-03.txt, October 1999.

[3] Jini home at http://www.sun.com/jini/.

[4] E-Speak home at http://www.e-speak.hp.com.

[5] Salutation home at http://www.salutation.org/.

[6] UDDI home at http://www.uddi.org/.

[7] Gulbrandsen, A. and Vixie, P., "A DNS RR for specifying the location of services," *IETF RFC 2052,* October, 1996.

[8] Gribble, Steven, et al., "The Ninja Architecture for Robust Internet-Scale Systems and Services," *Computer Networks Special Issue on Pervasive Computing*, Vol. 35, No. 4, March, 2001.

[9] Scott, Donna, "Commentary: More Hardware Won't Solve Web Site Outages," *Gartner Viewpoint*, January, 2001.

[10] Buyya, Rajkumar and Vazhkudai, Sudharshan, "Compute Power Market: Towards a Market-Oriented Grid," *Proceedings First IEEE International Conference on Cluster Computing and the Grid*, May, 2001.

[11] Kempf, James and Goldschmidt, Jason, "Notification and Subscription for SLP," *IETF Internet-Draft draft-kempf-srvloc-notify-05.txt*, January, 2001.

[12] IETF Middlebox Communication Working Group home at http://www.ietf.org/html.charters/midcom-charter.html.

[13] Yaffe, Joel, "Why Established Hosting Providers Should Fear Loudcloud." *Giga Information Group IdeaByte*, August, 2000.

[14] Schatt, Stan, "Enterprise Network Building Blocks: Crafting an E-Business Infrastructure," *Giga Information Group Planning Assumption*, March, 2000.

[15] Carpenter, Brian E., "Architectural Principles of the Internet," *IETF RFC 1958*, June 1996.

[16] Zhao, Weibin, et al., "mSLP – Mesh-enhanced Service Location Protocol," *IETF Internet-Draft draft-zhao-slp-da-interaction-09.txt*, October, 2000.

[17] de Silveira, Gledson Elias, and Fabio Q. B. da Silva, "A Configuration Distribution System for Heterogeneous Networks," *Proceedings Twelfth Systems Administration Conference (LISA 1998)*, December, 1998.

[18] Chandra, Prashant, et al., *Darwin: Customizable Resource Management for Value-Added Network Services*, November, 2000.

[19] Foster, Ian, Carl Kesselman, and Steven Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *International Journal of Supercomputer Applications*, 2001.

[20] Grimshaw, Andrew, Adam Ferrari, et al., "Legion: An Operating System for Wide-Area Computing," *IEEE Computer*, Vol. 32, Num. 5, May, 1999.

[21] Appleby, K., L. Fakhouri, et al., "Océano – SLA Based Management of a Computing Utility," *Proceedings IFIP/IEEE International Symposium on Integrated Network Management*, 2001.

[22] Wilkes, John, Patrick Goldsack, et al., "Eos – The Dawn of the Resource Economy," *Proceedings HotOS VIII*, May, 2001.

[23] OpenSLP home at http://www.openslp.org.

[24] Traugott, Steve and Joel Huddleston, "Bootstrapping an Infrastructure," *Proceedings Twelfth Systems Administration Conference (LISA 1998)*, December, 1998.

[25] Howes, Tim, "The String Representation of LDAP Search Filters," *RFC 2254*, December, 1997.

[26] National Institute of Standards and Technology, "Digital Signature Standard," *Technical Report NIST FIPS PUB 186*, U. S. Department of Commerce, May, 1994.

[27] Chase, Jeffrey B., Darrell C. Anderson, et al., "Managing Energy and Server Resources for a Hosting Center," *Proceedings Eighteenth Symposium on Operating System Principles (SOSP)*, October, 2001.

[28] Burgess, Mark and R. Ralston, "Distributed Resource Administration using cfengine," *Software – Practice and Experience*, Vol. 27, p. 1083, 1997.

TemplateTree II: The Post-Installation Setup Tool

Tobias Oetiker – ISG.EE, Swiss Federal Institute of Technology

ABSTRACT

After installing an OS distribution, a computer is generally not yet properly adapted to go into production at the local site. Security policies must be implemented, local services configured, and standard application settings deployed. Solutions to this problem range from unpacking a tar archive in the root directory to sophisticated tools like Cfengine [3].

This paper presents TemplateTree II, a highly modular approach for solving the post-installation problem which uses Cfengine as a transport mechanism.

Introduction

System management of an IT-infrastructure with more than just a few machines, requires an overall management concept and a lot of automation to preserve the sanity of all involved. This has been discussed at length, for example, in the Infrastructure Paper [1] by Traugott and Huddelston and and also a few years back by Rémy Evard at LISA 1997 [2].

One of the problems to tackle is the post installation procedure of freshly installed machines as well as feeding and care once they are in operation. This paper presents a modular solution to this problem. Another related problem is software distribution which is not covered in this paper (see [8] for our solution to this problem).

The Problem

Integrating a machine into the local environment requires that configuration files are replaced with customized versions for the local site, or even that whole new software is added to the machine. Files like /etc/services, /etc/inetd.conf, and /etc/mail/sendmail.cf come to mind, but also third party packages like AFS (Andrew File System), SSH (Secure Shell) or Postfix.

In principle this problem is quite easy to solve: Setup some sort of Master Server which holds a copy of all the changes you want to apply to a client and let the client update itself by copying all the changes from the Master Server. The Infrastructure Paper calls this machine the 'Gold Server.' Manage the material on the Master Server using CVS and you even have reproducibility and accountability.

Unfortunately, most often not all the clients are the same, so there must be a method to define what material from the Master Server should go to each client. A simple approach taken by many, is to write a special customization script which is run on each machine and figures out what should be done depending on information it gleans from the machine at runtime.

One tool which takes this approach to a new level is Mark Burgess' Cfengine [3]. Essentially it provides a highly specialized language for describing machine configurations, so instead of writing customization scripts in Perl or bash it is now possible to write them in a language created for this very purpose. I am comparing TemplateTree II mostly with Cfengine because it is the most widely used tool in this area.

At first we thought that Cfengine would solve all our problems, but unfortunately we found three main areas where it did not fit our requirements:

Modularity and meta configuration: Because our setup is quite diverse with many different configurations we wanted a system where we can build configurations based on individual components. Each component would expose certain configurable parameters which allow to tune the component to the setup it is going to be used in. One could talk about an additional abstraction layer. Cfengine has no concept for modularization apart from its ability to use include files. The include files can not define a configuration interface for themselves.

Documentation: The best system can be difficult to understand if no documentation goes along with it explaining the reasoning behind the less obvious configuration decisions. All you get in Cfengine configuration files are embedded comments.

Disk-less client support: We use disk-less clients whenever possible. Because all client filesystems are stored on the server we can update them even when the client is not running. Cfengine has been designed for running locally, this means that writing a Cfengine configuration file for our setup is more complex than necessary. Especially, there is no support to change the 'root' directory within a Cfengine configuration file.

Despite the missing features we found Cfengine to be a great tool for the task of actually doing the necessary modifications on the target machines. It can

serve both as a network transport and as an all dancing and singing file handling tool. We decided to implement a configuration system on top of Cfengine which outputs Cfengine configuration files. This allows us to use Cfengine as a back-end system without having the problems outlined above.

Concept of a Solution

TemplateTree II addresses all the problems mentioned above. It provides a post-install host configuration system based on *Feature-Packs*.

Modularity Through Feature-Packs

Modifying a freshly installed machine to fit the local requirements normally consist of several loosely coupled tasks. These tasks are, for example: linking the machine into the local user authentication system, configuring the machines mail transfer agent, adding the latest OpenSSH distribution and turning off unnecessary services. In the context of TemplateTree II these independent tasks are called *Feature-Packs*.

Feature-Packs are self-contained in the sense that you can mix and match Feature-Packs from a central repository. Several *system management domains*[1] could share a single repository. This is similar to the *independent packages* approach chosen for the SEPP software distribution System [8] or the Classes of Synctree [4] or even Cfengine configuration include files.

Splitting Configuration and Code

The modifications necessary to make a machine fit into the local setup may be similar across many machines. Nevertheless, some differences between machines will exist. If a single Feature-Pack should be able to cater for all these situations it must itself expose a configuration interface. For a mail server you want to be able to set the local mail domain, for an automounter setup the automount maps may differ between departments. In publishing, one of the hot topics is separating content from design [9]. In TemplateTree II we might call it separation of configuration from code.

Obviously this is not new. Most software packages support some sort of configuration file and you do not have to recompile emacs to change the size of a font. Therefore, in our case we might talk about a unified configuration level which sits above the normal application configuration files.

[1] A system management domain is a a set of machines managed by a single system management group. This can be a single server with a few clients or a complex setup with many servers and clients. Size does not and should not matter from a technical point of view. TemplateTree II will fit both. What we have here is more a political and organizational question. I avoided to use the word infrastructure, because its definition of encompassing all machines of a whole organization is not really appropriate in our institution with thousands of machines and many independent system management groups.

TemplateTree II implements such a meta configuration level. This has two advantages: First, a single Feature-Pack can be deployed across many different machines in various configurations. Second, the relevant meta configuration information is kept separate from the Feature-Packs and is therefore more manageable.

Collaboration

With TemplateTree II it is possible to maintain a central repository of Feature-Packs. A group of system managers can work together keeping them up to date. Each maintains a number of Feature-Packs in the central store, specializing in some areas. When it comes to defining what Feature-Packs to use in a certain system management domain, each of the participating managers has his full freedom, as to which Feature-Packs he wants to use and how he wants to configure them. This potential for collaboration is quite similar to SEPP [8].

Centralized Management

Having a way to easily customize machines is not enough. We also need to manage the configuration information in an efficient way. We wanted an efficient method for writing a single configuration file per management domain. A single configuration file for all machines is more efficient to maintain and has less redundancy than a system with large amounts of configuration data. Configuration files of other tools in the same problem space like Cfengine [3] tackle this problem by implementing whole scripting languages in their own right. In the case of Ressmans paper from LISA 2000 a SQL database [10] is holding all the necessary information. For TemplateTree II we chose a configuration file centering on which Feature-Packs to apply to which group of machines and how to configure the Feature-Packs. This gives us all the configuration freedom we need while still being quite simple because the complexity is locked away into the Feature-Packs, while the configuration information remains in the central configuration file.

Documentation

System management concepts and tools differ largely from site to site, so there is no official book for folks to read in order to get up to speed on working in our environment. To make sure the documentation gets written and updated as part of our daily routine, we tightly integrate facilities for documentation into all our system management tools (see [8]).

Being able to turn a machine from "freshly installed" into "useful workstation or server" in a short time, is nice. But this is only half the bill when either something fails or when several people work together on the task. The other half is having good documentation regarding the changes done to a machine. Not only do we want to know what has been changed but also why it has been done. TemplateTree II defines a mandatory documentation standard for all Feature-Packs. TemplateTree II integrates the documentation into the Feature-Pack itself. Following the

ideas of literate programming [6], it is possible to automatically create a big POD[2] file, documenting all the Feature-Packs. This means that when you want to use a Feature-Pack you will get full documentation about what the Feature-Pack does, how you can use it, and any special points to observe when applying it to your setup.

Disk-Less Clients or "No Magic Please!"

A major feature of Cfengine is, that you can write configuration files which react to the setup and current state of the local machine. In his Computer Immunology paper [5] Burgess uses this facility to illustrate how a self healing mechanism for computers could be implemented. Our setup contains many disk-less clients where we build the client filesystem on the server even before the client boots for the first time. This means TemplateTree II must be able to run without access to the machine it is customizing. Therefore all the information it needs is available in its configuration files.

Cfengine provides facilities for monitoring machines and even for reacting to certain problems. The scope of TemplateTree II is more focused.[3] Its only purpose is to modify the configuration of a machine to make it fit the local requirements. This task is completely controllable. No evaluation of the status of a certain machine is necessary in order for TemplateTree II to do its work. We know what machines we have and how they are configured.

If some configuration must be done locally and while the client is running, there is always the option of applying a specialized boot script to the client or to add an appropriate cron-job.

[2]POD is a very simple documentation format widely used in the Perl community. It can be converted into Man, HTML, LaTeX, and other formats. We use it for most of our technical documentation (see [12]).

[3]Monitoring the health of the system is left to a specialized tool in this area (Gossips [11]). Note that using TemplateTree II does not prevent you from using cfengine as an immunological agent. It only means that TemplateTree II will not make use of these aspects of cfengine.

Getting the Modifications to the Client

TemplateTree II uses Cfengine as a transport mechanism for moving and applying files to the target machine. We decided to use Cfengine because it provides all the file tackling equipment required for what we intend to do in one simple binary. It also allows us to use all the neat features of Cfengine like the Cfengine daemon or its ability to only copy those files which have changed or to do a dry-run in order to test a new configuration.

TemplateTree II outputs a single Cfengine configuration file per management domain. This configuration file contains all the information necessary for configuring each individual machine, as well as the root directories of the disk-less clients.

Architecture

Figure 1 shows the main components of TemplateTree II. The configuration is stored in three main configuration files:

system.conf defines where TemplateTree II should look for other components of the system, which operating systems your installation supports, and which attributes must be known for each host managed by the system. The system configuration does not have to be changed under normal circumstances and is therefore kept in its own file.

site.desc contains structured information about how hosts should be arranged into groups, which Feature-Packs should be installed on which group of hosts, and how each feature should be configured.

host.list holds a simple table with a set of basic attributes for each host. The Feature-Packs can draw upon this information in addition to specific 'per pack' configuration parameters.

The actual files which have to be applied are stored in a repository of *Feature-Packs*. A Feature-Pack is a directory containing all the files which must be applied plus a file called META which describes how to apply the files.

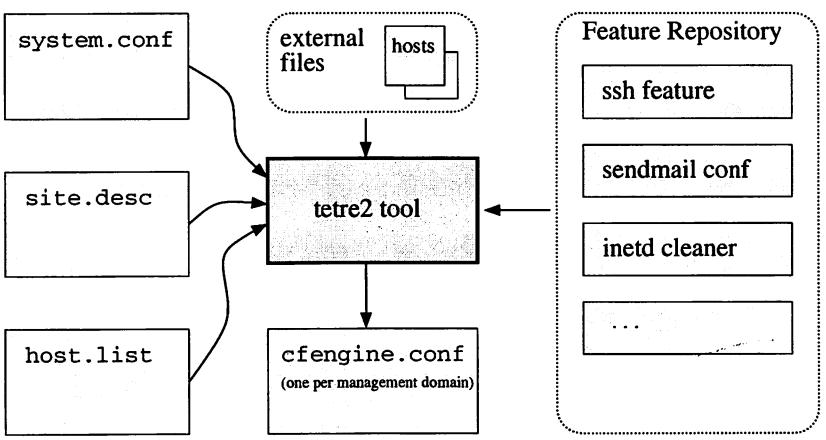


Figure 1: The components of TemplateTree II.

Based on the configuration files and information taken from the selected Feature-Packs, the tetre2 utility builds a cfengine.conf file.[4] The Cfengine configuration will contain references to files from the Feature-Packs repository if whole new files must be copied as part of the post-installation process. Apart from this, the generated Cfengine configuration will not depend on TemplateTree II anymore. Together with the Cfengine daemon it is possible to update the files on any machine regardless whether it shares a common filesystem with the machine where TemplateTree II is installed or not. It only needs the Cfengine binary to work and network access to the Cfengine daemon.

For large changeable files like /etc/hosts a special facility is provided to keep these files separate from the Feature-Packs. A Feature-Pack can in fact define that it wants to use such a file. TemplateTree II will then make sure that this file is provided.

Using TemplateTree II

Configuration Splitting

Probably the most complex task when starting to use TemplateTree II is to split your setup into a set of Feature-Packs. The goal is to devise a scheme which allows to use different combinations of your Feature-Packs to cater for all the special needs at your site.

Our approach is to use four different types of Feature-Packs:

Application: This is for essential applications like OpenSSH or Xinetd which are not part of the basic OS distribution, but which we think should be.

OS: Here we collect all the OS dependent changes we apply to every machine if it runs a certain OS release.

Domain: All machines within the same authentication domain usually share some files and configurations.

Machine: And finally there are always some bits and pieces of configuration which are special to an individual machine. For these we create individual, machine dependent, Feature-Packs.

The site.desc configuration file then defines which combination of Feature-Packs has to be applied to each machine.

Anatomy of a Feature-Pack

A Feature-Pack is a directory containing all the files necessary to implement a certain functionality or behavior on the target machine. For an OpenSSH Feature-Pack this would be all the binaries for all the architectures the Feature-Pack is going to support, a startup script, and configuration files defining the site policy.

[4]The architecture of the tetre2 application allows to quite easily add other output formats apart from Cfengine configuration files like shell scripts, for example. TemplateTree II already has the ability to output POD documentation on all the Feature-Packs available in the current repository.

```
:
+- features
|  |
|  +- sendmail_config-1.0-to
:  |  +- META
   |  +- sendmail.cf-client
   |  +- sendmail.cf-server
   |
   +- openssh-2.9.4-to
   :  +- META
      +- ...
```

Figure 2: A Sample Feature-Pack repository.

In addition to this free-form collection of files every Feature-Pack must contain a file called META. It describes the content of the Feature-Pack and contains all the instructions required to apply the Feature-Pack to a machine. Figure 2 shows the directory layout of a Feature-Pack repository. The META file offers six ways to expose configurable items from the Feature-Pack:

File Sets: The content of a Feature-Pack can be split into blocks. Each block must have a distinct name. In TemplateTree II such a block is called File Set. When applying a feature which uses File Sets, the administrator can choose which File Set to apply to which Machine.

Operating System: Template Tree knows for each machine it manages, which OS is running on that machine. A Feature-Pack can be configured to install different files depending on the release and type of OS it is running on.

Substitutes: Some files are the same for all machines except for one word, which has to be changed according to, e.g., the name of the default printer or the mail domain. A Feature-Pack can expose several named parameters which can be configured when using the Package.

Automatic Substitutes: The host.list file contains a number of key parameters for each machine. A Feature-Pack can access these parameters as search and replace data on files it contains.

Magic Substitutes: The value of a Substitute can be subjected to some perl manipulation prior to being used in a search and replace operation.

External Files: A package can "contain" external files. The Feature-Package only knows how to apply these external files to the target machine. The name and location of these files is configurable in the site.desc file when using the Feature-Pack.

The next page shows a sample META file. It shows how to use the functions listed above in context.

The `*** Action ***` section of this META file is very boring as it only contains `copy` instructions. TemplateTree II supports a number of other actions for creating directories, removing files and directories, generating symbolic links and, finally, a special function for assembling files.

```
*** Name ***
Sendmail Config Package

*** Version ***
1.0

*** Maintainer ***
Tobias Oetiker <oetiker@ee.ethz.ch>

*** One Line Description ***
Configure Sendmail to work properly

*** Blurb ***
This package can configure sendmail as either a normal mail server or
as a null client simply feeding mail to a central mail server.

*** Usage Info ***
This package assumes that the stock sendmail version for your OS is
already installed. It does not contain any binaries, just an
appropriate configuration file.

*** File Sets ***
client     A configuration for a forward only client
server     The works, a full blown server with built-in jacuzzi

*** Change Log ***
2000/07/02  to  Demo package created
2000/08/10  to  Added change log

*** OS Support ***
sol26
sol8
rhl62

*** Substitutes ***
mdomain  Name of the local mail domain.
mserver  Host Name of our mail relay

*** External Files ***
aliases     Your site's alias file

*** Action ***

# the server gets an alias file. Which physical file
# gets copied to the server can be configured when setting
# up the Feature-Pack in the site.desc file.

server:.*:
C aliases      /etc/mail/aliases        644  root:root

# solaris mail servers get to use the file sendmail.sol.server as
# their sendmail.cf file. While copying the file to the server,
# cfengine will do a search/replace operation for >#>mdomain<#<
# and >#>mserver<#< according to the setup in the site.desc file

server:sol.*:
C sm.sol.srv /etc/mail/sendmail.cf 644  root:root   mdomain,mserver

# the same happens for RedHat mailservers except that there is a
# different sendmail cf file.

server:rhl.*:
C sm.rhl.srv /etc/mail/sendmail.cf 644  root:root   mdomain,mserver

# for null clients we do not need a os sepcific sendmail.cf file, so
# the OS part of the file selector is set to a globally matching
# regular expression.

client:.*:
C sm.clnt   /etc/mail/sendmail.cf 644  root:root mdomain,mserver
```

Listing 1: A sample META file.

The *assemble* function allows for different Feature-Packs to each provide part of a file which then gets *assembled* on the target machine. This can be used to configure /etc/system on a Solaris system where some Feature-Pack might want to add a special driver load instruction whereas on other machines there is just the usual shared memory and stack protection configuration in there. Another usage would be the root crontab file or the inetd.conf where several Feature-Packs contribute to the contents of the file.

The Host List (host.list)

The most simple configuration file in a TemplateTree II setup is the host.list file which contains a simple table with all the hosts of the site. It is shown in Listing 2. The third row in the sample above is for the disk-less machine called *bluehat* which uses disk-space on the server *drwho*. Some of the columns in the table like the host name and the OS are required by TemplateTree II, others are configurable through the system.conf file.

The System Configuration (system.conf)

At the root of the TemplateTree II configuration setup is the system.conf file. It defines where the other components of the system are stored, what OSes are handled and which columns must be listed in the host.list file.

```
*** Locations ***
SiteDesc = /etc/tetre2/site.desc
HostList = /etc/tetre2/host.list
Features = /etc/tetre2/features
ExternalFiles = /etc/tetre2/extfiles
ConfServer = jobis.ee.ethz.ch
RunTimeVar = /var/cfengine

*** Operating Systems ***
sol26    Sun Solaris 2.6 Sparc
sol7     Sun Solaris 7 Sparc
sol8     Sun Solaris 8 Sparc
rhl62    RedHat Linux 6.2 x86
irix63   SGI Irix 6.3 MIPS

*** Host List Config ***
HOST     Hostname
IP       IP Address
ETHER    ETHERNET Address
DEF_GW   Default Gateway
DOMAIN   DNS Domain of the Host
OS       OS of the machine
ROOT     Where is the ROOT of this machine

*** Host List Tests ***
DOMAIN   sub {return \
     "We only manage ethz domains" \
     unless $Match =~ /ethz/; 0 }
```

The Site Description (site.desc)

With all the other parts of the system in place it is now possible to setup the Site configuration file defining which Feature-Pack should be installed on which machine and how it should be configured in the process. The site.desc file has three main sections:

Feature Selection: This allows to create instances of Feature-Packs. Each instance can contain several *cases*. A case lends its name to a group of configurable parameters. When applying a Feature-Pack these parameters can be activated by using the name of a *case*.

Host Grouping: Most hosts at a site share some common configuration. So instead of configuring each host individually you can build groups of hosts and then apply the Feature-Pack instances to the groups.

Feature Application: In the final section of the site.desc file the feature instances are applied to individual hosts or whole groups of hosts as defined in the previous section.

Below is a tiny Site Description sample file for illustration:

```
*** Feature Selection ***
#-----------------------------------
SENDMAIL := sendmail_conf-1.0-to
#-----------------------------------

  # default values
  mdomain = "ee.ethz.ch"
  mserver = "smtp.ee.ethz.ch"
  aliases -> "aliases/ee"

  # case 'mailserver' uses
  # the file-set 'server'
  mailserver:
        /server

  # case 'nullclient' uses
  # the file-set 'client'
  nullclient:
        /client

*** Host Groups ***

null = drwho bluehat

*** Host Features ***

tardis: SENDMAIL(mailserver)
@null: SENDMAIL(nullclient)
```

Security Considerations

If a whole site is setup and configured using a centralized approach such as TemplateTree II the potential problems which arise are similar to those you get when all your fields carry the same crop. First, a

```
# HOST   IP            ROOT                          OS     DOMAIN
#-------------------------------------------------------------------
tardis   192.168.1.2   /                             Sol8   ee.ethz.ch
drwho    192.168.2.44  /                             Sol7   ee.ethz.ch
bluehat  192.168.2.12  drwho:/export/root/bluehat    Sol7   ee.ethz.ch
...
```

Listing 2: Table of hosts: host.list file.

security problem present on one host is likely to be present on all and second, if your central configuration machine gets compromised and the intruders modify the TemplateTree II setup then the malicious code could get distributed easily to all managed machines.

We counter this by two measures: First, we keep the whole TemplateTree II setup in CVS which allows us to backtrack configuration problems we introduce ourself and, second, we protect the central TemplateTree II configuration server by only allowing access via secure channels. Further, the whole system gets backed up regularly and is trip-wired for easy detection of unauthorized modifications. We have also been thinking about using digital signatures on FeaturePacks but have not yet implemented such a functionality.

Conclusion

By building on top of Cfengine a system has been devised that allows for the complete modularization of the post install process of Unix workstations. TemplateTree II allows us to perform fully customized machine setups in a very short time while maintaining full reproducibility. Because TemplateTree II works with a single top level configuration file the configuration information for the whole site is readily accessible and when changes are required, they can be quickly performed on all systems.

Future Work

At the moment we are happy with TemplateTree II and use it as it stands. We do have some ideas, though. One would be to improve the Cfengine configuration generator to create more compact Cfengine configuration code. Currently the Cfengine configuration is about as voluminous as it can get. Smaller code would be simpler to understand and debug than the bulk we have today.

An entirely different road would be to replace Cfengine altogether and use a modified rsync server which provides a view on a virtual file system which changes its *contents* dynamically depending on the host which is sending the request. This approach would allow us to capitalize on the excellent performance and security possible with rsync/ssh.

About the Author

Tobias Oetiker is a Senior System Manager with the IT Support Group of the Department of Information Technology and Electrical Engineering at the Swiss Federal Institute of Technology in Zurich. He is an electrical engineer by education and a system manager by vocation. His main area of interest is currently scalable system management concepts and their implementation. In his spare time Tobi likes to read, go to movies, and work on his Free (as in GNU) software projects.

Acknowledgments

I would like to thank the following individuals for helping me with this paper: Paul Anderson, David Schweikert, Jon Stearley, and Fritz Zaucker. If you find anything especially well written or spelled properly, it must have been one of their suggestions.

Availability

TemplateTree II is written in Perl and available under GNU GPL from http://isg.ee.ethz.ch/tools/ .

References

[1] Steve Traugott, Joel Huddleston, "Bootstrapping an Infrastructure," http://www.infrastructures.org, *LISA*, 1998.

[2] Evard, Rémy, "An Analysis of UNIX System Configuration," *LISA*, 1997.

[3] Burgess, Mark, "Cfengine: A Site Configuration Engine," *USENIX Computing Systems,* Vol. 8, No. 3, http://www.iu.hioslo.no/cfengine, 1995.

[4] Lockard, John, Jason Larke, "Synctree for Single Point Installation, Upgrades, and OS Patches," *LISA 1998.*

[5] Burgess, Mark, "Computer Immunology," *LISA 1998.*

[6] Donald E. Knuth, *The CWEB System of Structured Documentation*, Addison-Wesley, 1993.

[7] Weisshaus, Melissa, et al., *GNU tar: An Archiver Tool*, http://www.gnu.org/manual/tar/ .

[8] Oetiker, Tobias, "SEPP – Software Sharing and Packaging System," http://www.sepp.ee.ethz.ch/, *LISA 1998.*

[9] Lamport, Leslie, "LaTeX: A Document Preparation System, User's Guide and Reference manual," Addison-Wesley.

[10] Ressmann, David & John Valdés, "Use of Cfengine for Automated, Multi-Platform Software and Patch Distribution," *LISA*, 2000.

[11] Goetsch, Victor, Albert Wuersch, Tobias Oetiker, *Gossips: The Systems and Services Monitor*, http://isg.ee.ethz.ch/tools .

[12] Wall, Larry, Tom Christiansen, "Perl POD: Plain Old Documentation," http://www.cpan.org/doc/manual/html/pod/perlpod.html .

The Arusha Project: A Framework for Collaborative Unix System Administration

Matt Holgate – University of Glasgow
Will Partain – Arusha Project

ABSTRACT

The Arusha Project is an independent open-source project centered on the premise that the best hope for Unix system administration at modest-sized sites is through large-scale Internet-wide collaboration. We present a simple *object model* as a thinking tool, and an XML-based *configuration language* as a concrete notation for expressing system-administrative facts. We show how this framework allows a gentle evolution from current practices, but gets us quickly to very powerful ways of working.

Introduction

Fifteen years ago, people would stare at you blankly had you suggested a major operating system might be written by a few hundred people scattered around the world in their spare time (more or less).

Today, people stare at you blankly if you suggest that many thousands of Unix system administrators[1] might manage their sites *collaboratively*, with no organization to 'set standards' and no single concept of what 'good' administration is. Welcome to the central idea of the Arusha Project!

The Arusha Project (ARK[2]) is an independent 'itch-needing-scratched' open-source effort that sprung up in Glasgow, Scotland, in 1998. Whether it will see its vision of rampant collaboration fulfilled is a mostly sociological matter.[3]

For system-administrative collaboration beyond a well-tuned mailing-list, a more formal *lingua franca* (a 'trade language' used between people of diverse mother tongues) is essential. The core ARK technology, its *configuration language*, is our contribution to this end. The ideas behind it are innovative, and its design is a delicate brew of minimalist choices.

This paper outlines the context that we care about, followed by a sneak preview of our final results. We then set out some key engineering choices, plus a word or two about the wider collaboration picture. The ARK configuration language is the main technical content, followed by a review of what we gain, collaboratively speaking.

The View From Mount Meru[4]

The Arusha Project is hugely informed by its target context, so it is worth explaining that in some detail.

Picture: To your left you see a pile of just-delivered Unix workstations (and servers, and networking bits, and ...), and to your right you see a marauding herd of users with work to do. You, the glorious administrator, are to assemble the pile of boxes to your left into the most important tool of the mob to your right, helping them toward staggering, competition-wilting job effectiveness.

Now move the clock forward by ten years. Most of those original computers have been retired, replaced spasmodically by others (budget gods willing). That known-to-work-together pile of equipment has turned into a made-to-work-together hodge-podge. Duct tape has been used. People, too, have come and gone.

The administrator's goal will not have changed. The amalgam of equipment should still comprise a single 'tool' that is perfectly tuned for its users' effectiveness. (Better than that, actually: a good system will continuously *anticipate* the workplace's needs a year or two ahead.) All with great uptime, no unplanned outages, perfect backups, and apple pie for all.

Our picture provides a formidable administrative challenge in the context we see: 4-400 hosts, with a comparable number of users in an innovation-oriented workplace.[5] Some employers (e.g., investment banks) may spend enough money to collect a surfeit of heavy-hitting administrators. Other employers may be wise or lucky enough to hire clever LISA readers. But, all too often, you will find one-or-a-few under-resourced administrators doing their best but not winning. If you hear the phrase, "We spend all our time fire-fighting," then you have found our target audience.

Our beleaguered fire-fighters will be doubly frustrated if they are cursed by wanting to aim high:

[1] Hereafter, just 'administrators' (and 'administration,' 'administrative,' ...).

[2] 'ARK' is Arusha's airport code (*Arusha* airport, not Kilimanjaro International, JRO). We use 'ARK' and 'the Arusha Project' interchangeably.

[3] *Peopleware* argues convincingly that 'technology' and 'process' are always less important that 'people' issues in a technical endeavor [DeM87].

[4] Arusha is a town in northern Tanzania that sits at the foot of Mt. Meru, which is 14,979 feet high, the fifth highest mountain in Africa. What better place for a general overview?

[5] Such a workplace benefits from a rich and arguably over-provisioned computing environment, with considerable information flow and exchange.

wanting to build a wonderful system. (And is it fun to do anything else?)

We now outline some of the main problems (labeled with **P-*n***) and goals (**G-*n***) that face our overeager fire-fighters. (The labels are so we can refer back to them.)

P-1 *Complexity is unavoidable:* No matter how you go about it, 4-400 Unix hosts, well stitched together, will make for a *complex* system; and managing complexity well is hard work.

P-2 *Mediocrity is natural:* A modern Unix system is composed of dozens, if not hundreds, of subsystems (kernel, DNS, sendmail, RAID, printing, ...), each of which is a complicated beast. You can often make a Good Living Indeed by being seriously good at *just one* of these subsystems. An administrator who spends a morning with one of these subsystems has to accept that s/he is probably mediocre at it.

P-3 *Wayward buses are a threat:* It is all too easy for essential site knowledge to live *only* in the head of one administrator, but if s/he is run over by the proverbial bus ...

P-4 *Isolation:* In our target context, an administrator often works alone, or a small team works at a level far below 'critical mass.'[6] Solutions, scripts, documents, etc., are *unlikely* to receive any independent scrutiny; this is *not* a recipe for robust and powerful systems.

P-5 *Wheel reinvention:* This tendency is fairly natural in an isolated systems ecology, and human hubris tends not to help. Also, the site-specific nature of scripts that administrators write weakens their value as reusable components.

P-6 *Administration is more than fiddling with* /etc*:* Just as there is more to software development than design and coding, there is more to administering a Unix system than managing hosts, users, and software applications.

Administrators have lots of other 'things' to manage: vendor records, maintenance contracts, old purchase orders, licenses, serial numbers, spare-parts inventory, helpline numbers and addresses, e-mail about all of the above, and so on.

One may choose to have no formal management of such 'things,' with the attendant risks. However, as soon as you turn such information into bits to be cared for, you have entropy (or 'bit rot') to worry about. Phone numbers change, new releases come out, the purchasing department 'improves' their procedures, etc.

P-7 *Systems have a multi-decade lifespan.* Computing infrastructures tend to be evolving, long-lived structures. It is not rocket science to spend lots of money this year and have a great system for users *this year*. What *is* a real achievement is to keep entropy at bay for ten or twenty years and still have a sweet system running, budget vagaries and turnover vicissitudes notwithstanding.

P-8 *Administrators are not, by and large, programming-enthralled.* Administrators admire a good Perl script as much as the next guy, and are more than willing to roll up their sleeves and sling a little code. But tell them that they 'only' need to grasp (say) Hindley-Milner type inference [Mil78] to make some Great Leap Forward, and they will (mostly) respond, "No thanks."

P-9 *Working examples to learn from are rare.* Many administrative tasks, e.g., setting up a mailing-list server, are an amalgam of small tasks. The README file typically says, "You could do this, or perhaps that." What you really want to know is what did someone, anyone, *actually do* to complete the job. Even better would be several real, working, non-obsolete examples to study and work from.

G-1 *Administration should be 'site-at-a-time.'* If your administrative gestures (running a command, clicking a GUI button, ...) have small effects, e.g., adding a single line to a file on one host, then it takes many gestures to do the job (low productivity). We want high-impact gestures that make something true for the *whole site*. But more important than automated ways of doing things is *thinking about* the system in 'site-at-a-time' ways. (This is the same impulse as the 'infrastructures' work [Tra98].)

G-2 *All added-value matters.* Recall our initial picture of just-delivered boxes to the left, users to the right ... We consider *all 'value-adding' activities* that lead to a useful-to-users system to be 'system administration' and therefore within the Arusha Project domain.

Setting /etc/resolv.conf correctly for all hosts is in play (nothing new there ...), but so is the hpux-admin article about kernel tuning that you saved. So are notes of a phone call to Sun support. All are part of the total 'added value' that makes the system what it is.[7]

This goal says that we need something analogous to the 'methodologies' of the software domain. The observation there is that software development is much more than just frenetic coding, and all of the many other artifacts and activities that go into making software (requirements gathering, unit testing, project plans,

[6] We define a team at 'critical mass' as having enough collective brainpower to get on top of the problems immediately at hand, and to have enough cycles left over to engage with the wider world (e.g., read LISA proceedings) and to do some speculative projects, to explore new technologies, etc.

[7] Aside: this paper concentrates on what we do *to the boxes* on the left in order to induce great work in the user population. Actually, we consider it equally within an administrator's remit to Do Things *to users* (training, cajoling, altered work practices, new ways of thinking, etc.), if that makes the total result better.

etc.) need to be 'managed' and dealt with inside some overall 'process.'[8]

G-3 *Simplicity.* In this line of work, simplicity of design and implementation is invariably rewarded.

G-4 *A system should have its 'source code.'* We want as much of administrators' 'added value' to be expressed as bits-on-disk as possible, if only to avoid the Wayward Bus problem (**P-3**). We view the bits created directly by administrators (scripts, notes, e-mail, web pages, etc.) to be the *source code* for the system.

Of course, the idea of 'source code' implies something stronger: with nothing but the source code and raw vendor-supplied hardware/software, the overall system should be entirely reproducible.

G-5 *Once and Once Only.* Just as duplicate code is suspect in programming, a system's source code should have the same "once and once only" property.[9] If a site has a hub-and-clients sendmail configuration, that should be expressed in *one* piece of 'source,' and *not* in files scattered hither and yon. It is good to look in a single directory and be able to say, "That is all there is to know about our sendmail setup"; it is downright perverse to cut-and-paste sendmail.cf fragments from host to host. That is malignant source-code duplication.

And if saying something once per site is good, once per planet is even better.

G-6 *It's great for sites to be different.* A common impulse in administration is to 'standardize' (hardware, software, people, ...), nearly always as a way to cut costs. The notion is not entirely without merit.

'Standardizing' often bumps into an organization's *Immovable Local Realities* (ILRs). These are less-than-ideal local facts or components that simply must be factored into any system solution. Examples might include: poor cable ducts, a cantankerous old plotter that is essential to the enterprise, some key software whose supplier has long-since gone out of business. ILRs often guarantee that administrators must deal with 'not as standard as we'd hoped' solutions.

Another worry with 'standardizing' is that it increases the risks associated with single-vendor solutions. This year's 'obviously the way to go' can crumble if your vendor loses a few key people, or stumbles into an unfortunate lawsuit. A deeper problem with 'standardizing' is that it is often at the expense of *competitive advantage*. If you are committed to making your users more effective than the competition, you will have to supply something extra, something different that the other guys cannot just order off the web. Decreeing "A Windows PC for everybody!" is not an option that your competition somehow stupidly overlooked.

G-7 *Presentation matters:* While there have been many Unix sites that were rigorously managed, there have been many fewer where this was manifestly obvious to someone other than the administrator who did the work.

[8] Actually, we are profoundly skeptical about the conventional 'processes' and 'methodologies' of the software world. But you are going to have an administrative 'methodology,' whether you call it that or not; so you might as well try for a good one.

[9] "Once and once only," usually written 'OAOO,' is a buzzphrase of Extreme Programming [Bec99].

Cairo[10]

We have established a setting for ARK, and will shortly describe the ARK configuration language and 'engine.' But, you cry, "Cut to the chase scene! What might I do with this stuff at my site?" Here are some deeply hypothetical examples, barely explained.

Build all packages for all hosts (even if of diverse platforms):

```
ark package install ALL
```

Verify the configuration of all Solaris hosts:

```
ark host verify sparc-solaris
```

Check that local mailing lists only have valid subscribers:

```
ark maillist chk-valid local-lists
```

Any support contracts about to expire?

```
ark support-contract list-expiring ALL
```

Notice how *many kinds* of administrative 'added value' are being managed in a consistent way.

Crucially, we want these (and many other) powerful 'site-at-a-time' commands to incorporate *both* local ways of working *and* top-quality 'patterns' from respected ARK sites around the world.

We hasten to add that this kind of power does not fall ready-made out of an ARK tarball. You must build up an ARK description of your site; but this local effort to make such powerful commands possible is modest. At one of our real sites, an average package (file) clocks in at 20 lines (690 bytes), a host at 32 lines (1140 bytes), and a 'disk chunk' [think of an automount-map entry ...] at nine lines (295 bytes). Often, even these small files can be copied from a collaborating pal.

Volcanic Ash[11]

Our context and goals make for a big picture. Happily, the Arusha Project is *not* about filling in the whole picture; rather, we provide a *framework* within

[10] Arusha lies about halfway along the Cape-to-Cairo overland route, which is the direction most people do it. Cairo is their ultimate goal.

[11] The ground beneath your feet in Arusha is a powdery greyish volcanic ash; we hope the foundations of the Arusha Project, sketched in this section, are more solid.

which cooperating administrators world-wide can set about painting the canvas. Still, even a framework must make some up-front (engineering) choices that affect its shape and scope.

We first review a few simple pragmatic choices, noting how they tie into our problems and goals.

Small sites: Our target is comparatively small sites (4-400 hosts), and scaling issues do not keep us awake at night. Lots of people target the single-host site, and others are better placed to tackle the Big Sites ('enterprises').

High on the food chain (i.e., existing tools): We strongly prefer to build on existing tools (notably the standard Unix utilities), especially for the heavy-lifting parts of administrative tasks (**P-5**).

For scripting, administrators should still be able to use shell/Perl/Python. They *don't* have to learn a new language, throw out their old stuff and start again (**P-8**).

Textual tools must suffice: Administrators sometimes work with the world crashing around them. They must not be forced to rely on an elaborate software scaffold to get any work done.

Our *central choice* is unsurprisingly:

Internet-wide large-scale collaboration: We simply cannot see any other way for administrators at small sites to produce top-quality results. (**P-4**)

Other basic design choices follow from our collaboration imperative.

Separate mechanism and methods: Our 'it's OK to be different' goal (**G-6**) means that the core ARK machinery needs to provide a mechanism that doesn't constrain the methods used in administration. Sites ought to be able to pursue different policies, architectures, and/or methods, yet still be able to express their solutions in an ARK framework.

Reuse must be lavishly supported: Good ways to reuse administrative solutions (across organizational boundaries) are essential; otherwise, sharing to overcome administrator isolation (**P-4**) clashes with the need for site-specific solutions (**G-6**).

Not all-or-nothing: ARK must not be an all-or-nothing proposition. If the only way to 'get into' the Arusha Way is to start building a site over again, administrators will (rightly) walk on by.

Object orientation: We need some extra thinking machinery. For this, we steal a simple form of 'object-oriented' thinking from the software world. It is ubiquitous, universal in application, and is essentially a complexity-managing tool. (**P-1**)

Using XML as the main notation: We need some extra notational machinery. For this, we make simple use of XML [XML]. It has the merit of being a standard, and of perhaps being faintly familiar to administrators. The tidal wave of XML-related tools should make it easier to get presentation right as well. (**G-7**)

XML's extensibility means that different sites can use different tags to encode their unique information. And the way XML can wrap around other programming text means that the 'business end' of a solution can use a specialized tool (e.g., PIKT [Ost]) or remain in a familiar scripting language (**P-8**).

What we value most about XML is its *semi-structured* nature: the level of precision of a description can range from utterly precise (very structured) to exceptionally loose (unstructured) ... For example, here is an install method that is precise:

```
<install><code>
cd /build/dir/foo
/usr/bin/make install
</code></install>
```

And here is a much looser 'equivalent':

```
<install><code lang="message">
You will need to be in the directory
where foo is built.  Typing
'make install' should do it.
</code></install>
```

XML lets us express all of our 'added value' (**G-2**) but without forcing us to do so rigorously. ARK can then support an evolution from informal solutions (probably text) to precise ones (probably code). (**P-7**)

All of that said, we are *not* that excited about XML.

Tribal Matters

The Arusha Project (ARK) is substantially a social enterprise, and that is not the subject of this paper. But we do need to mention 'teams' and comment upon the old 'but we cannot give away our secrets' chestnut.

Teams

The *team* is the basic social unit in ARK. All code is tied to some team. A *site team* produces and manages the code specific to a site. A *methods team* produces and manages code that is site-independent and which (presumably) promotes particular ways of doing administrative things; the team's members might share an office, or be scattered around the world. And finally, there is at least one *mechanism team*: the base team, also called 'ARK,' provides the ARK engine.

How teams go about their business (what they promote, how they distribute their bits) is entirely up to them. Teams may have profoundly different notions of administration. A healthy Arusha world would comprise many site teams, possibly just the one mechanism team, and perhaps four or five general (how-to-run-a-site) 'methods' teams. There might also be quite a few specialized teams that target a specific domain, e.g., how to run a particular flavor of website.

Collaboration vs. Competition

The ARK collaboration imperative may seem at odds with a goal of seeking competitive advantage through better computing infrastructure.

There *are* some aspects of administration you would not reveal to a competitor: your exact mix of tools, your /etc/hosts.allow file, anything related to your core competence, and so on.

But it is simply delusional to think that your standard-issue Apache configuration (say) somehow represents a rival-clobbering breakthrough. On the contrary, if the configuration was set up by a solo administrator without any peer review (**P-4**), the quicker you expose it so that a fellow ARKer can knock off the rough edges, the better.

Strangely, we have a hunch that Internet-wide administrative collaboration may work better than that within an organization. Inside a company, interactions (between, say, 'satellite' and 'corporate' IT groups) may be clouded by issues of funding, status, or political advantage. Meanwhile, the stranger in Croatia who critiques your Big Brother setup is probably a disinterested party.

Even if publicizing considerable administrative work meets with a collaborative deafening silence, there is still one *certain* beneficiary, and that is the provider: we all do better work if we know others will see it.

Swahili[12]

The single unavoidable piece of Arusha technology is the ARK *configuration language* and the *engine* that interprets it.

Things, Fields, Values (and More)

The ARK configuration language has an XML syntax and can describe any entity an administrator cares about. The following figure shows (contrived) examples of the unavoidable entities (or *things*, in ARK-speak) – teams, hosts, and packages.[13] A team:

```
<team name="glasli1">
<contacts><list>
  <item>partain@dcs.gla.ac.uk</item>
</list></contacts>
<admin-group>sliadmin</admin-group>
</team>
```

A host:

```
<host name="slicker">
<status>active</status>
<ip-address>130.209.242.51</ip-address>
<history><doc format="html"><![CDATA[
<ul>
<li>2001.09.04: Rebooted (matt).
<li>2001.03.10: Add 2nd disk drive.
</ul>
]]></doc></history>
</host>
```

A package:

```
<package name="textutils--2.0.11">
<status>revealed</status>
<hosts-supported><list>
  <item>sparc-solaris7</item>
</list></hosts-supported>
</package>
```

All instances of an ARK 'thing' (e.g., all hosts) are said to be of the same *type*.

Besides the unavoidable types (teams, hosts, packages), a site may choose to manage other things: users, support call-outs, disks, network ports, ... the ARK engine is domain-agnostic. Some fictional examples of such things might be:

```
<user name="matt">
<status>active</status>
<cron-allow-fragment>
  <constraint>
    <host-spec>slinger</host-spec>
  </constraint>
  <string>matt</string>
</cron-allow-fragment>
</user>
```

```
<support-contract name="hp">
<terms>next day</terms>
<phone>+44 800 555 4321</phone>
<email>none</email>
<expires>2002.03.31</expires>
</support-contract>
```

Every thing has zero or more *fields*. For example, the <support-contract> example above has an <expires> field. Fields are *not* nested.

An individual field, whatever thing it is part of, has a structure drawn from a *fixed* set of *elements* (**G-3**). (That is worth saying again: *All* fields of *all* things have the *same* internal structure!) The most important field elements are:

A value: A *value* can be one of: a string, a list, a table (key-value pairs), a documentation fragment (various formats), or some code (Bourne shell, Python, or Perl). Preceding examples show at least one of each. A value can be nested in obvious ways, e.g., a list of tables of lists of lists of strings ...

Parameters: Values, most notably *code*, can be parameterized; this is a key reuse weapon, and essential to "once and once only" source code (**G-5**). So, for example:

```
<do-it-now>
  <param name="ECHO">/bin/echo</param>
  <code>$ECHO "I'm doing it now"</code>
</do-it-now>
```

Parameters usually have defaults, as in this example.

Constraints: A field's *constraints* guard its value; if the constraints are not satisfied (or cannot be made so), then the value/parameter-settings/etc. do not apply. Consider:

[12] Swahili is the *lingua franca* (trade language) of eastern Africa. Though it is mother tongue of some people who live on the coast, most Arusha dwellers would speak another language at home and use Swahili in civic life.

[13] All XML examples are slightly simplified compared to Real Life.

```
<used-for>
  <constraint>
    <host-spec>freebsd</host-spec>
  </constraint>
  <string>Web serving</string>
</used-for>
<used-for>
  <constraint>
    <host-spec>rhlinux</host-spec>
  </constraint>
  <string>SETI@Home</string>
</used-for>
```

If we ask for the <used-for> field in the context of a freebsd host, we get one string; if an rhlinux host, we get another.

Constraints are normally intra-type; for example, one package method depends on another. But constraints can also be cross-type; for example, a host method could depend on a user's attribute or a vendor's method.

Prototypes

You should now have some notion of how you might ARKishly describe all of your hosts (for example). For each machine, you would prepare an XML file (<host> ... </host>), each of which might have many fields, e.g., <ip-address>, <serial-num>, <disk-config>, <os>, etc.

This task would be hugely repetitive, because many machines would have the same elements for the same fields. The solution is to create a *proto-host* (a specific form of *proto-thing*) that expresses the common knowledge; e.g., see Listing 1.

Now, all the hosts that have these properties (presumably all 'lab machines') can have this (proto-)host as a *prototype*; for example:

```
<host name="slibber">
<prototypes>
  <prototype team="ours"
             name="lab-machine">
</prototypes>
<ip-address>192.10.168.4</ip-address>
<tagline>
  <param name="what">workstation</param>
</tagline>
</host>
```

'Things' as Objects

The 'prototypes' idea is drawn from the world of *classless* object-oriented programming languages [Bor86]. By 'object,' we mean an opaque entity that presents an interface to the world through public *attributes* (data about it which you can query) and *methods* (code that makes it 'do something').

The idea of a prototype-based object is incredibly simple: you create a new object by first copying another, and then tweaking the new object to make it unique. The object, or objects, that you copy (from) are the *prototypes* for the new object.

This is exactly what we are doing with ARK things. In the example above, the host named slibber is created by first cloning the prototype host lab-machine, and then adding/overriding-with its own unique data, notably the field <ip-address>.

An ARK thing (with prototypes) fits our definition of 'object,' above. A field with a non-code value is an attribute, and a field with a code value is a method.

For example, we can query our host object slibber for its attribute ip-address (a string), or for disk-config (a table). Or we can invoke one of its methods, perhaps restart-sendmail (implemented with a shell script).

A thing can have zero or more prototypes. A proto-thing can itself have prototypes. Exactly how this works out is explained later.

Inheritance

In our example above, the host slibber *inherits* the value its disk-config field from the lab-machine proto-host. Inheritance is a fundamental property of objects which helps to control complexity: we push common/reusable attributes and methods into the base (prototype) objects, which make them widely available to their inheritees. **(P-1, G-5)**

Combining this inheritance with parameterized values gives easy reuse of administrative solutions.

In ARK, field fragments are the *only* inherited entities. Thanks to our classless prototypes mechanism, there is no complex type/class structure to worry about as well **(G-3)**.

```
<host name="lab-machine" prototype="yes">
  <tagline>
    <param name="what" default="no" />
    <string>A @param:what@ in Lab 4</string>
  </tagline>
  <disk-config><table>
    <entry name="boot">  Quantum 9902</entry>
    <entry name="other"> IBM   4/4432</entry>
  </table></disk-config>
  <os>NetBSD 1.4.3</os>
  <restart-sendmail><code>
  kill -HUP `/bin/cat /var/run/sendmail.pid`
  </code></restart-sendmail>
</host>
```

Listing 1: A *proto-host* expressing common knowledge.

This style of 'value inheritance' is close to that in Couch's Babble [Cou00] and that recently explored by Anderson [And00b].

Textual and Semantic Entities

A quick review: what do we type (textual), and what do we think about (semantic)?

A 'thing' is a semantic entity to which we apply 'object' thinking. The total textual material that comprises that 'thing' comes from potentially-many files, one per thing or inherited proto-thing. In our example, two files: slibber.xml and lab-machine.xml.

Similarly, a 'field' is a semantic 'atom,' the smallest piece of a 'thing' that we can get a hold of. The total textual material that comprises a 'field' is one or more like-named *field fragments*, plus the prototype links that connect them. In our slibber example, its <ip-address> field comes from one field fragment. However, its tagline field comes from two fragments, the *partial* fragment (no value) in slibber.xml and the completing fragment in lab-machine.xml.

Crucial point: even the smallest entity in ARK land, a field, can be built up collaboratively by different teams scattered around the world!

Other Language Complexities?

We have glossed over some details of the configuration language, mostly unexciting details of what you can do with a <code> value. We are sometimes asked for further details about 'modules,' or include files, or DTDs, or other suspected language facilities; in short, 'there must be more to it' ...

No, *that is all there is* (**G-3**). (For a complete rundown on field elements, see the language manual [ARK].)

Prototypes As Matching and Naming Mechanisms

A prototype name can be a *pattern* against which we match 'real' (non-prototype) things. When we had a <host-spec>freebsd</host-spec> constraint, it simply meant that any real host which has the freebsd prototype as one of its prototypes will match.

Similarly, *naming* a prototype thing is equivalent to naming all of the real things that have that as a prototype.

Prototypes as Cross-Planet Inheritance

Notice that every <prototype> 'link' must specify a *team* ('.' is shorthand for the prevailing site-team).

This team can be (and ideally will be) a global ARK team. By using the prototype things it supplies, you take advantage of others' expertise that may benefit your site's requirements. You also maximize the pool of people who will be interested in (and critique) your own contributions.

We like the idea that people around the world work to improve our systems while we are asleep.

The Operational View

How do we write code to *use* the ARK objects (things), to access their attributes and invoke their methods? In Python, you write code that looks like this:

```
# create a host "object" for
# our 'slibber':
slibber = ark.host.ArkHostsMgr(). \
            lookup('slibber')

# access and print out its IP#:
print slibber.ip_address()

# run its restart-sendmail method:
slibber.restart_sendmail()
```

This is object-oriented programming at its simplest. What lies behind these rather lovely method calls is the ARK *engine*, which scrambles over <prototype> 'links' to create the illusion of the built-up-by-copying objects.

The prototype links in an ARK 'thing' comprise a directed acyclic graph; the nodes visited by doing a depth-first left-to-right traversal give a thing its *prototype path*.

Operationally, the ARK engine walks along a prototype path. This has the same semantics as actually copying proto-things, but is more efficient.

The executive summary version of the algorithm is: walk the prototype path looking for the field of interest, checking constraints and collecting parameters as you go, returning immediately when you find a field fragment with a value. (For tables, we keep going and 'merge' all of the entries found.)

The Command-Line View

We provide a command-line tool ark, with which we can 'fire' common methods for ARK objects. The syntax is: ark *type method [options] [thing1 [thing2...]]*.

The examples given earlier should now make sense!

Big Game Spotted[14]

We have described enough of the ARK configuration language to show why it is effective 'glue' to hold together world-wide collaborative administrative effort, as well as the merits of our simple object model. We note connections to the problems (**P-*n***) and goals (**G-*n***) of the first section.

Collaborative Wins

Domain- and methods-agnostic: The ARK configuration language has virtually no system administration wired into it. Choosing to express solutions with it in no way compromises local ways of working.

Optional: Managing aspects of your site with the ARK machinery is optional. If you don't want <vendor>s as part of your ARK world, then don't.

Evolutionary: You can start small with ARK, and grow bigger (**P-7**). Also, you can bring non-ARK

[14] Arusha is awash in safari operators, as it is the 'jumping-off point' to all of northern Tanzania's game parks. Most tourists hope to see the Big Five: buffalo, elephant, leopard, lion, and rhino.

bits of your world into ARK play quite easily. Consider this variant on the method we saw earlier:

```
<restart-sendmail><code>
  /usr/local/sbin/restart-sendmail
  </code></restart-sendmail>
```

(We presume the referenced script is pre-ARK.)

Lingua franca, not mother tongue: Though we envisage a strange administrative world of 'objects,' 'methods,' etc., we expect this to materialize mostly by straightforward 'wrappers' around conventional solutions. The 'wrapping' is this funny ARK stuff, but the 'business end' is still your favorite scripting language (**P-8**), or Cfengine, Nessus, RPM, Swatch, or any other tool of your choice.

Simple pragmatics: ARK uses a dull form of XML, and requires little more than a text editor and a Python interpreter. Most fields in most XML files are a few lines long, particularly for site teams, which typically inherit most of their smarts from a global team. (**G-5**)

Simple mental model: The ARK prototype-based 'object' model is as simple as they come, but packs a heavy punch (**G-3**). It means we can think about *all* aspects of a site's administration in a *uniform* framework.

Simple data: Strings, lists, tables, bits of scripting or documents... not a lot to get your head around.

Rich reuse: The basic way that a field become reusable is by putting it into a prototype thing. As a field is pushed up a prototype tree, it applies to more and more inheriting 'objects.'

If we then parameterize some aspects of inheritable attributes/methods, their reusability is considerably enhanced. Our experience is that you do this slowly, as the need arises.

A small unit of collaboration, the 'field': Collaboration does not work if the parties have to agree on too much up front. The smallest 'atom' of ARK collaboration is a single field of a particular type of thing. Sites can do quite different things with ARK <host>s, but if they can agree on just a few fields (e.g., <ip-address>, <gateway>, and <dns-servers>), useful collaboration can follow.

ARK is easy to learn: Our experience is that a competent Unix administrator can learn everything they need to know about the ARK configuration language in half a day.

Object Wins

Our dominant mental metaphor, of 'objects' with attributes and methods, is a powerful (and well-known) way of thinking about systems. It attracts benefits of its own.

Universal: The ARK configuration language can describe *any* aspect of a system that you care about. (**P-6**, **G-2**)

Should you choose to record a preponderance of administrative 'added value' with the ARK language, you then have comprehensive 'source code' for your system (**G-4**), which is a key defense against administrators falling under buses (**P-3**).

If many sites record their solutions in the ARK language and make them available to others, then we have a ready source of complete, accurate, probably-automated examples that administrators can study (**P-9**).

Abstracted away from immediate file contents: Because ARK is about 'objects' rather than files and bytes, it tends to operate at a high level of abstraction. 'Site-at-a-time' operations (**G-1**) are the most obvious manifestation of this; for example configuration files that require a different format for each platform can be produced from a single object representation.

Complexity control: The fundamental strength of object thinking is managing complexity. We try to push complexity 'upward' into a prototype, so that many things can inherit from it (spreading the complexity cost over more things) (**P-1**).

If we push the complexity into a *global* team, we hope to find ourselves working with other clever people (in the relevant team) to keep the fancy stuff right. "With enough eyeballs, all bugs are shallow" (Linus's Law, according to Raymond [Ray00]). This is how you can beat the inherent mediocrity (**P-2**) in a small isolated administrator team (**P-4**).

Will They Bite?

So, will administrators working in our context bite and do the ARK thing? As we have said, we are sure the answer is more sociology than technology. One of the biggest obstacles today is IT management that optimizes for 'cost' and has never thought of optimizing for 'effective.'

Administrators will need to learn enough XML to talk the ARK talk; they probably know a little HTML already, so it is not a stretch.

Administrators really should know how to write Bourne shell scripts, ARK or not. Perl and Python are optional.

Administrators are slow to adopt *any* site-configuration tool, because they (rightly) know that it is a decision that will be hard to escape. We've tried to make ARK suitable even for glacially-paced incremental adoption.

Related Work

There is a flotilla of tools that offer comprehensive management of a single host: HP's SAM, IBM's SMIT, Linuxconf, and many others. These are all nearly useless in our context.

The literature (and world) is crawling with 'site configuration' systems; Evard's 1997 paper [Eva97] is

a particularly useful review. On the specific matter of configuration *languages* (notations), Paul Anderson's survey [And00b] is a good overview of the territory. He draws on the venerable Edinburgh work on LCFG [And00a], as well as Cern's SUE [SUE], Cfengine [Bur] and others. If you venture over to the world of software deployment, there are *lots* of related things; a good starting point is the work by Hall, et al. [Hal98]. Moving not much further afield and you reach the bewildering land of 'software configuration management' ... [App]

Our ARK work tends to differ from other 'site configuration' tools in that they make no upfront provision for collaboration across organizational boundaries. Couch's DISTR system [Cou97] is an exception, with collaborative concerns very similar to ours, but limited to file distribution. Another system in a comparable vein is 'PowerAdmin' [Pow], a customizable service that diverse groups within the University of Michigan can 'buy.'

There are many systems where a difference in scale is apparent. In the high-performance computing arena, there is much effort (and money being spent) on *computational grids*, e.g., the Globus Project [Glo], aspects of which are squarely in our domain, notably the Metacomputing Directory Service (MDS). All-inclusive commercial administrative tools, e.g., CA's Unicenter TNG [CA], HP's OpenView [HP] and Tivoli's tools [Tiv] seek to cover the wide ground that ARK does; again, however, such solutions tend to be well beyond the means of our target audience (and often platform-specific).

In his analysis, Evard [Eva97] suggests that the "systems administration community needs stronger abstraction models." We believe the ARK object model contributes here. As mentioned before, both Anderson [And00b] and Couch [Cou00] have dabbled with a similar form of 'value inheritance.' We further note that the FLASH project in Brazil also picks up on object-oriented ideas, in a more complex way [daS98].

There are a few other systems that try to capture constraints among configuration artifacts, as we do. Ganymede [Abb98] is one example: it is a directory service into which 'local smarts' can be programmed. The work by Couch and Gilfix with Prolog [Cou99] is another powerful (and scary) way to tackle such constraints. Again, if you move slightly further afield, you find many comparable systems; one example is the CML2 Linux-kernel configurator [Ray01].

Roads From the Arusha Clock Tower[15]

The Arusha Project (ARK) is not about producing a tool; rather, it hopes to be at the center of a maelstrom of collaborative system administration. The core developers' activity is driven by 'scratching the itches' at their real sites, which may or may not have value to others. We would hope that most Arusha activity will evolve to happen well beyond our purview.

One aspect of ARK that we expect to occupy us for a while is the presentation (or documentation) of a system (**G-7**). We think of it as bringing the literate-programming impulse [Knu84] to the system source code (as represented by the ARK XML files). (We have a first-cut implementation, using the Webware application server [Est01], also written in Python.)

The ARK configuration language is *domain-agnostic*, and so the question arises: in what other fields might it reasonably be applied? For example, the 'chipmake' tool, for describing how to put together a semi-custom chip, was scarily close to ARK in the issues that it had to address [Hol00].

Summary

The Arusha Project (ARK) has a profoundly ambitious goal of many thousands of Unix sites around the world being managed in a collaborative way.

We began this paper with a from-first-principles analysis of our target context, reaching a set of goals we would have for any system-administrative framework. The ARK configuration language provides a basis for meeting all of those goals. It is an XML-notated *lingua franca* with which system administrators can describe the value they have added to a collection of raw vendor-supplied computing equipment. Their descriptions are in terms of *objects* ('things'), with parameterized *attributes* ('fields') and *methods* (fields with code values). We have a *universal* and *as-precise-as-you-like* language for describing administrative activities (because of the semi-structured nature of XML).

Our object 'things' are built up out of *prototype* objects, possibly supplied by other teams, potentially anywhere in the world. Administrators *collaborate* on shared solutions insofar as they use (and work to maintain) common prototype objects.

The ARK 'object' view of administration is lightweight, builds on standard Unix tools, and allows extremely varied uses, from do-just-one-specific-task to run-the-whole-site.

The following ideas are unique to the Arusha Project:

- Describing *all* administrative 'added value' in a *single* notation (the ARK configuration language
- Viewing these descriptions as *objects* which link together *across* organizational boundaries;
- Advocating *world-wide collaboration* as the basic way forward for Unix system administration.

[15] The Arusha town center features a clock tower, next to which is a signpost giving distances to other locations. What with Arusha's place along the Cape-to-Cairo route, the places listed tend to the remote: London, Moscow, Cape Town, etc.

Acknowledgments

The Arusha Project (ARK) is an independent open-source project that has been based in the Computing Science Department at Glasgow University. We have received financial support for LISA-related expenses from the Department (including some under SHEFC RDG Project 85, "Design Cluster for System Level Integration"), and from Verilab (http://www.verilab.com). We are very grateful to all concerned.

ARK work has origins in the glamake tool (1997), which automatically built open-source software for multiple platforms. The earliest (undistributed) ARK code was written in Haskell [Has]. ARK was set up as a SourceForge project in January, 2000. (Hooray for SourceForge!)

A small army made this paper hugely better than what we started with, including our LISA reviewers and shepherds, and also David Partain, Jonathan Hogg, Rolf Neugebauer, and Tommy Kelly. Thanks, folks.

Status and Availability

The central Arusha Project website is http://ark.sf.net/. It includes instructions for getting any and all ARK bits.

We recommend that prospective ARK users join one or more of the ARK mailing lists. Also, the website recommends some get-your-feet-wet activities to try before you switch to an ARK lifestyle.

At time of writing, there are a few real production Unix sites fully managed in the Arusha Way, mostly in the chip-design ('technical computing') arena.

Author Information

Matt Holgate is a Research Assistant on the IDEAS project, which is funded by SHEFC RDG 130. He is based in the Computing Science Department at Glasgow University, Scotland. He is also a part-time system administrator for Verilab, a Scottish hardware design and verification company. He graduated with an honours degree in Computer Science from Trinity Hall, University of Cambridge in 1998. Contact: matt@dcs.gla.ac.uk.

Will Partain is a graduate of Arusha School, Rift Valley Academy, Rice University (BSEE), and University of North Carolina at Chapel Hill (Ph.D., computer science). He was one of the original development team for the Glasgow Haskell Compiler, at the Computing Science Department, Glasgow University. Contact: partain@dcs.gla.ac.uk.

References

[Abb98] Abbey, Jonathan and Michael Mulvaney, "Ganymede: An Extensible and Customizable Directory Management Framework," *LISA 1998*, Boston.

[And00a] Anderson, Paul, and Alastair Scobie, "Large Scale Linux Configuration with LCFG," http://www.dcs.ed.ac.uk/home/paul/publications/ALS2000/.

[And00b] Anderson, Paul, "A Declarative Approach to the Specification of Large-Scale System Configurations," version 1.9, Discussion Document, 2001, http://www.dcs.ed.ac.uk/home/paul/publications/conflang/.

[App] Appleton, Brad, "The ACME Project: Assembling Configuration Management Environments (for Software Development)," http://www.enteract.com/~bradapp/acme/.

[ARK] "The ARK configuration language manual," http://ark.sf.net/ark-conflang.html.

[Bec99] Beck, Kent, *Extreme Programming Explained: Embracing Change*, Addison-Wesley, 1999,

[Bor86] Borning, A. H. "Classes versus Prototypes in Object-Oriented Languages," *Proceedings of the ACM/IEEE Fall Joint Computer Conference*, pp. 36-40, 1986.

[Bur] Burgess, Mark, *Cfengine*, tool, http://www.iu.hio.no/cfengine/.

[CA] Computer Associates, *Unicenter TNG*, tool, http://www.cai.com/unicenter/.

[Cou97] Couch, Alva L., "Chaos Out of Order: A Simple, Scalable File Distribution Facility for 'Intentionally Heterogeneous' Networks," *LISA 1997*, San Diego.

[Cou99] Couch, Alva and Michael Gilfix, "It's Elementary, Dear Watson: Applying Logic Programming To Convergent System Management Processes," *LISA 1999*, Seattle.

[Cou00] Couch, Alva L., "An Expectant Chat about Script Maturity," *LISA 2000*, New Orleans.

[daS98] da Silva, Fabio Q. B., Juliana Silva da Cunha, Danielle M. Franklin, Luciana S. Varejão and Rosalie Belian, "An NFS Configuration Management System and its Underlying Object-Oriented Model," *LISA 1998*, Boston.

[DeM87] DeMarco, Tom and Timothy Lister, *Peopleware: Productive Projects and Teams*, Dorset House Publishing Co., 1987.

[Est01] Esterbrook, Chuck, "Introduction to Webware for Python, 9th International Python Conference," Long Beach, California, March, 2001, http://webware.sf.net/Papers/IntroToWebware.html.

[Eva97] Evard, Rémy, "An Analysis of UNIX System Configuration," *LISA 1997*, San Diego.

[Glo] "The Globus Project," http://www.globus.org.

[Hal98] Hall, R. S., D. Heimbigner, and A. L. Wolf, "Evaluating Software Deployment Languages and Schema," *Proceedings of the International Conference on Software Maintenance*, November, 1998; See http://www.cs.colorado.edu/serl/cm/Papers.html.

[Has] "A Short Introduction to Haskell," http://www.haskell.org/aboutHaskell.html.

[Hol00] Holgate, M. and J. Hogg, "Chipmake: An XML-based Distributed Chip Build Tool, 1st ECOOP Workshop on XML and Object Technology," *Sophia Antipolis*, France, June 12, 2000.

[HP] Hewlett-Packard, *OpenView,* tool, http://www.openview.hp.com/.

[Knu84] Knuth, D. E., "Literate Programming," *Computer Journal,* Vol. 27, No. 2, pp. 97-111, 1984.

[Mil78] Milner, R., "A Theory of Type Polymorphism in Programming," *Journal of Computer and System Sciences*, Vol. 17, pp. 348-375, 1978.

[Ost] Osterlund, Robert, *PIKT (Problem Informant/Killer Tool)*, tool, http://pikt.uchicago.edu/pikt/.

[Pow] "The ITD PowerAdmin service," http://www.umich.edu/~gpcc/poweradmin/.

[Ray00] Raymond, Eric, "The Cathedral and the Bazaar," http://tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/, 2000.

[Ray01] Raymond, Eric, "The CML2 Language: Constraint-based Configuration for the Linux Kernel and Elsewhere," http://tuxedo.org/~esr/cml2/cml2-paper.html, 2001.

[SUE] "SUE (Standard Unix Environment)," tool, http://wwwinfo.cern.ch/pdp/.

[Tiv] Tivoli enterprise management tools, http://www.tivoli.com/.

[Tra98] Traugott, Steve and Joel Huddleston, "Bootstrapping an Infrastructure," originally published in the *Proceedings of the Twelfth USENIX Systems Administration (LISA) Conference*, Boston, Massachusetts, 1998, http://www.infrastructures.org/papers/bootstrap/bootstrap.html.

[XML] "Extensible Markup Language (XML)," http://www.w3.org/XML/.

Lexis EXam Invigilation System

Mike Wyer and Susan Eisenbach – Imperial College

ABSTRACT

Computers have made their way into the classroom and lecture hall. Overhead projectors, blackboards, and whiteboards are being displaced by smartboards and computer based multimedia presentations. Students with laptops are a common sight and many courses have their lecture notes on the web. Students are studying programming, web-site design, computer graphics, and many other practical disciplines, yet these courses are still being assessed with traditional pen and paper examinations.

When the Computing Department of Imperial College decided that their programming courses would be assessed with a computer-based paperless exam using our standard Linux [8] workstations, we were asked to make the labs secure enough to take an official exam. Here we present the issues and technologies involved in securing Linux for this purpose, and the software we developed to administer our examinations.

Introduction

People learn to program by sitting in front of a machine and typing. However, formal programming examinations are usually hand written on paper. So the skill being tested is not the same as the one being learned.

At Imperial College, we have had years of experience in running low-priority, low-security programming tests on the standard lab systems. These tests consist of a few simple programming questions, with the students expected to code their answers within the allotted time, submitting via an automated email-based system. Given the small amount of credit available and suitable vigilance on the part of the test coordinator, it was felt that these tests did not warrant additional security measures on the workstations.

Students and staff preferred the computer based tests to traditional written papers – the students felt much more comfortable programming in an editor with the chance to run their code, and the staff were able to compile and run the submitted code directly, which reduced the burden of checking the syntax and the correct solutions of the given problem by hand. In addition, the perennial problem of reading handwriting was removed.

Our regulations are such that one of the necessary conditions for passing the programming course is that a student must pass the final examination. With the popularity of the programming tests, we were asked to investigate the feasibility of running examinations securely on the lab systems. We were given the task of configuring our lab machines in such a way that students could safely take an official University of London examination on them.

At the time, our computing labs consisted of over 200 PCs, ranging from 233 MHz Pentiums to 800 MHz Athlons, all running RedHat Linux [11].

Requirements

Although most people are familiar with the security arrangements which accompany an official examination, they are not often encountered in a systems administration context.

These requirements are taken from the specification document discussed and agreed by the Academic Committee in the Computing Department of Imperial College.

Aims

Provide:

- familiar lab-like environment during exams,
- all resources necessary to complete exam,
- secure environment for completing exam,
- secure means of collecting exam answers.

Ensure:

- no access to unauthorized data,
- no access to other users on network,
- no distraction or interference from other users on network.

Further Details

Some exams may involve providing students with templates, stub code fragments, or other data. Likewise, the student will be required to create or modify files as part of the exam. The students will not have access to shared network volumes, so any files needed for the exam will need to be provided by the software examination system. Some standard applications need to be available.

Each completed exam submission must be securely stored and associated with the right candidate number. Exam submissions must not be accessible to anyone except the authorized agents of the University.

As with any other examination, students will only be allowed access to permitted resources. In addition to the usual physical precautions of a written exam (no books, paper, phones, radios, tattoos, etc.),

the student should not have access to unauthorized data. All access must be removed from:

1. data previously stored on hard disk in a writable area,
2. data on removable media (floppy or zip disk),
3. data on network device (home directory, bit-bucket)
4. communication via network.

It is also important to make sure that other users on the network do not interfere with the student during the exam, the on-line equivalent of the noisy mob in the corridor outside an examination.

Investigation

Development time was limited, so it was important to investigate currently available solutions. Several commercial products exist, for example WebCT [17] and Blackboard [2], but they are windows based and only offer support for traditional style exams. Indeed, a paper by Braun and Crable [3] strongly suggests in-house development as an alternative to the existing tools.

Although no existing package provided all the facilities we needed, there was a good chance that some of the individual tasks could be covered by one of the many security tools, packages, and utilities available for Linux [8] . The project to build a system to help administer examinations was dubbed Lexis, Lab EXam Invigilation System.

Network Access

A way of severely restricting the network was needed, and the most obvious and effective method would be to simply disconnect the network during any exam. Our network topology and hardware are such that this is a fairly straightforward option. The target machines would then be required to function correctly without any network. This raised several concerns about reliability, synchronization, and monitoring.

What would happen if a machine had a fatal problem during the exam, say a hard disk head crash? How long would it take to recover any data, if it was possible at all? These issues encouraged us to look at other solutions to the problem. Leaving the network connected also introduces problems. There were still reliability issues, cheating might be easier and the whole exam could be open to external attack.

It was vital that the worst-case failure of any of the constituent systems would not invalidate the exam. In order for Lexis to be a success, the safety, security, and reliability of pen and paper had to be matched.

We investigated Linux kernel level firewalling as an alternative to complete network disconnection. Linux 2.2 was the stable kernel at the time, so the ipchains interface, was evaluated [6] . The evaluation proved to be very positive, since ipchains provided us with a mechanism for filtering IP packets so that we could implement our firewall.

Using ipchains would give us precise control over the network traffic to and from each workstation involved in the exam. While this is not a novel idea to anyone who has been using ipchains, the key factor is that using ipchains provides an easy way to achieve *temporary* network security while still allowing certain connections. The "certain connections" we had in mind were specifically OpenSSH [14] connections to a central server. For brevity, we refer to OpenSSH as ssh.

Most firewalling schemes are permanent; with Lexis, the rules are in place for a few hours. Not only do the rules have to be automatically applied, they have to be removed as well. While the techniques involved are straightforward, the implementation must be fast and absolutely reliable.

System Security

We needed a strategy to prevent cheating – access to unauthorized data, tools, or other users.

Many UNIX systems use the chroot system call to restrict processes to a limited "sandbox" environment. This works very well for daemons which have a specific function and whose resource requirements (libraries, device files) are known in advance. In order to provide a similar setup for an exam, we would be forced to replicate a large percentage of the existing filesystem so that candidates would have access to the X Window System, window managers, all the editors and compilers needed for the exam, and so on.

Not only would all this file copying take a long time, it would take up more disk space than was available, and it it is not obvious that security would have actually been improved.

A similar strategy would be to dedicate a partition on the disk to Lexis, and dual-boot to specially configured OS and filesystem. As before disk space would be limited, and this approach has other drawbacks: we would have to provide compatible versions of the programming languages needed for the exam, along with having to provide a file transfer, security, and monitoring system. Although the security aspect would be simpler, we would still have to manage installation of up to three operating systems on the machine (Linux, Windows, and LexisOS, whatever that turned out to be).

We also considered creating a root filesystem image on the network which all the clients could mount, but this brought several more problems: using NFS (version 2) is not a good way to increase security, and where would the candidate's files be stored? If we wanted to use the local disk, we would still be stuck with the problem of sanitizing the filesystem and preventing the use of data or programs stored on that disk.

It seemed that no matter which approach we took we would need to come up with a simple, practical, and general way to secure Linux in a systematic

fashion. And if we could do that, then why not just run the exam from our newly-secured Linux environment which already had all the tools and configuration necessary to run lab software?

We started to analyze the types of activity that would be considered "cheating." It turns out that many of the activities which constitute illegal behaviour by students are privileged operations on the system. Operations such as mounting disks and creating trusted network sockets require either root access or set-uid root file permissions. By remounting the root filesystem without set-uid bits active, we eliminate the danger from setuid binaries. This cuts the risk from existing exploits of setuid code, and provides protection from trojans (e.g., a suid shell installed before the exam).

A useful side effect of this operation is that some system binaries that are installed setuid root (notably man and ssh) are also disabled. This would prevent the student logged into the machine from using the ssh client to attack the only open network channel (the ssh link to the Lexis server).

Reliability

One of the key concerns of the academics involved in the development of Lexis was that of reliability: what would happen if a PC crashed during the exam? While we could think of many analogous situations for a traditional paper-based exam which would be equally catastrophic, we wanted to show that a PC-based solution could improve upon the security and reliability of pen and paper.

To provide some protection from hardware failure, we decided that all client machines would dump the exam answers to a central server on a regular basis. This would provide flexibility to cope with any situation that might arise – if the candidate accidentally deleted important files, we would be able to restore them (at the request of the examiner); if a candidate disagreed with the marking of the exam and claimed that Lexis was responsible, we would be able to provide a detailed history of that candidate's work during the exam; if a machine failed, we would be able to restore the last dump to a different machine and let the candidate continue with minimal disruption. The main aim was to be able to support any decision made by the examiners.

It was also important to disable rebooting out of the provided secure environment. This, along with our other constraints was solved by our high level design decision to use of runlevel 4.

Runlevel 4

Runlevels are a standard feature of SysV-style init. Runlevels 0, 1, and 6 are reserved, and levels 2, 3, and 5 have (thanks to LSB[9]) fairly standard definitions across distributions. Runlevel 4 is available for use on many Linux systems. By using a runlevel specifically for lexis, we can use init [5] to handle transitions into and out of exam state, as well as providing a secure boot when an exam is in progress.

To start an exam, we create a new set of config files for the system, then change to runlevel 4. On changing runlevel, init stops services from the last runlevel and starts services for the new runlevel. We create a Lexis service that only runs in runlevel 4 that carries out any changes that need to be done on starting an exam or booting during an exam, including signalling the server that the workstation is ready for use and turning on the firewall rules.

This approach places most of the management burden on standard system processes, rather than on bespoke Lexis code. Unfortunately, the init supplied with RedHat 6.2 proved extremely unreliable during initial testing, often changing runlevel without running stop or start scripts. This meant that a large amount of the functionality of init (stopping services, restarting them) would have to be replicated in Lexis to ensure reliability.[1]

Exam files

To make the dumping of exam answers easier, we decided to restrict the candidates to a specific area of their local disk. /exam would be used to contain all the exam files and be the working area of the candidate. We only expect one candidate to use each machine, but any candidate could conceivably sit at any machine. We settled on the idea of a common home directory since this would mean we would only have to create the files needed for the exam once, and we would create special Lexis accounts that would only be valid for the duration of the exam. All the Lexis accounts would be in a 'lexis' group which would have access to /exam .

Special Lexis accounts would be necessary for several reasons:

- Our site uses Kerberos [7] , which relies on network access for authentication, so candidates would not be able to log in during the exam.
- According to University Examination regulations, candidates must only be identified by a candidate number. Using normal logins would compromise the candidates' confidentiality.

On our systems, this would necessitate disabling kerberos access, and providing new local Lexis accounts with appropriate passwords. Since physical access to the machines would be controlled by the usual exam invigilators, and we would need some way of associating candidate number with submitted files, we decided to make the username and password the candidate number. This would provide a double check at login that the candidate was using the right candidate number, and that all files owned by the candidate would be tagged with their candidate number.

[1] We have discovered to our cost that it is much easier for us to re-implement rather than trying to get Red Hat Software Inc. [11] to fix their product or accept patches from us.

Design

We decided on a client/server architecture, where the workstations that the candidates will use are the clients, and a central machine which monitors the exam and stores submitted answers from the candidates is the server. The overall structure of a Lexis session is summarized in Figure 1, showing how each client is individually firewalled to the server, and the points at which various illegal activities are stopped.

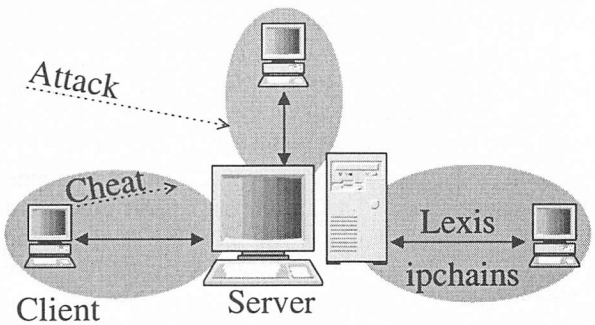


FIgure 1: Lexis architecture.

The Lexis protocol is very simple. All communication is in ASCII over an ssh link. All commands consist of a single line (terminated with a single newline). When the client is invoked by the server, the server sends its version number. If the client version matches, the client returns 'ok'. If the client and server versions do not match, or the client is not being run as the root user, an error is returned instead. For all subsequent commands, the client will return 'ok' if the call succeeds (after any expected output) or an error message if it fails. All error messages include the hostname of the client.

File transfer is accomplished using base64 encoding to make binary data safe to send over the ASCII link and MD5 checksumming [18] to ensure data integrity. This ensures the clients get the files they are supposed to from the server, and to make sure that the server receives valid dumps from the client.

Lexis Client

The main goal of the client software was to keep it safe and simple. The client files would have to be distributed to the clients ahead of time, and it would be extremely difficult (or even impossible) to make changes to the client during an exam. So the client software would have to provide the capability to cope with any situation that might occur during an exam.

We made the decision to use ssh to connect the server to the client. This would provide a simple STDIN/STDOUT communications channel between the server and client, as well as the means to get full remote shell access on the client from the server, to fix any problems remotely.

There would be just one program that communicated with the server (with others to accomplish specific tasks as necessary), and it would receive commands from the server and respond to them. At no point should the client be sending unsolicited data to the server. This meant that there would be no need to compromise the server by trying to enable the server to trust the clients.

Lexis Server

With the server, we wanted a straightforward system to manage connections with the clients, send and receive files, and respond to commands from the administrator. Since the clients would have limited functionality, most of the data processing would be done on the server, such as working out who had logged into which machine.

Software

Our client-server approach has the individual workstations as clients, with one or more central servers to communicate with the clients. The client software consists of three programs: lexis_startup, which is called by init [5] when switching to runlevel 4 (either at the start of an exam or on booting during an exam); lexis_active which is called by sshd [14] when a connection is made by the server; and lexis_warning, which is a simple X program that warns existing users that an exam is about to start. The Lexis session is managed on the server by a single process, lexis_server. The dump files stored on the server can be queried using the lexis_who and lexis_extract scripts.

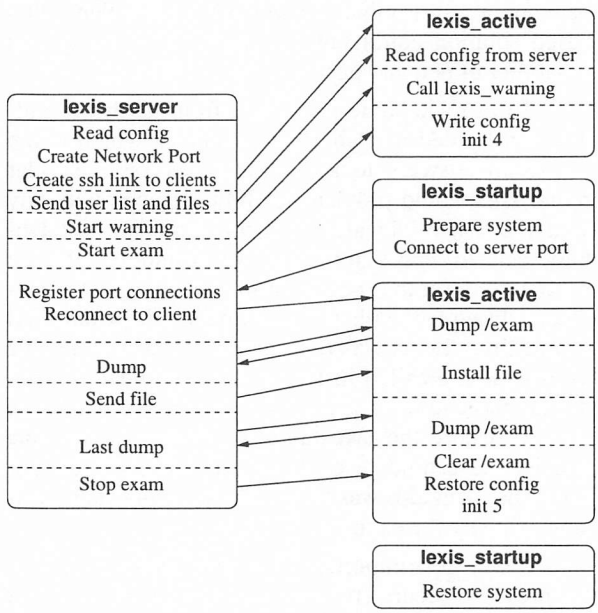


Figure 2: Main Lexis components.

The interactions between the main scripts – lexis_server, lexis_active and lexis_startup – are shown in Figure 2.

lexis_active

Most of the client-side code is in lexis_active, such as the file transfer mechanism, authentication setup, ipchains configuration, and runlevel control. It is

a straightforward perl [10] script which reads commands on stdin and produces output on stdout, and contains just over 400 lines of real code. It is designed to be invoked as a root process at the remote end of an ssh connection, and will abort if the calling uid is not zero. The commands are summarized in Figure 3.

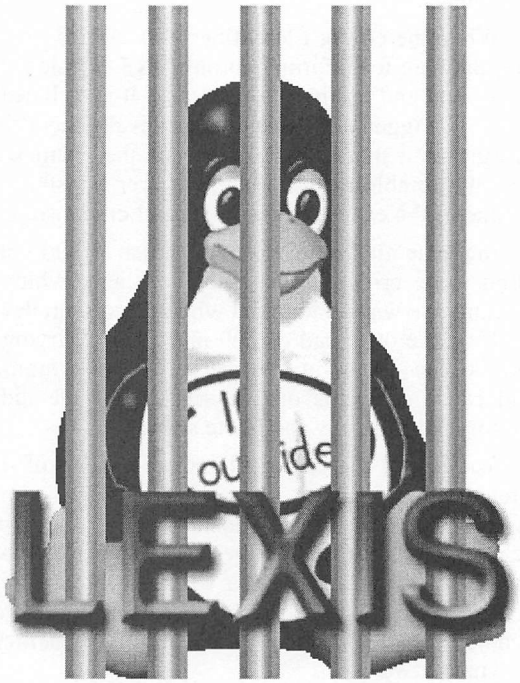


Figure 4: Lexis logo.

lexis_startup

One-off operations at the start and end of the exam are performed by lexis_startup, which is a SysV-style initscript. It is called by init when changing to runlevel 4 or when booting in runlevel 4. In either case, lexis_startup remounts the root filesystem with SUID bits turned off, clears tmp directories, shuts down non-Lexis services, redirects any remote syslogging to a local file (we don't want the system to lock up trying to contact a host its own firewall rules are blocking), opens a connection to the Lexis server, and updates the X display manager (gdm or kdm).

Terminating Lexis changes out of runlevel 4, remounts the root filesystem with suid bits set, reenables remote syslogging, and restores the X setup.

The display manager update is very simple, but necessary: we install a new logo to make it obvious the machine is ready for taking an exam, and restart X since it's /tmp/ lock-file has been removed, and it will automatically log out any existing users. The logo we use (Figure 4) is an adaptation of the classic Linux mascot, Tux, and shows him behind bars. The writing on his chest is "IC Outside," a logo we apply to the systems we build in Imperial College Computing Department.

There is scope for more paranoia in lexis_startup. The original idea was to recurse through the entire directory structure looking for world- or group-writable directories and clearing them. This strategy proved unworkable when we found that a

Command	Description
init	Clear */exam* and make the machine ready for use in an exam.
add server	Add the given IP address to the firewall rules and the list of hosts to contact when booting.
delete server	Remove the give IP address from firewall rules and the list of hosts to contact when booting.
port	Connect to the given port on the servers when booting.
rootpw	Set the root password for the current session.
user	Add the given username as a candidate.
users	Add the given list of whitespace separated usernames as candidates.
file	Transfer the given file to the client. If the filename is a relative path, transfer to */exam*, otherwise treat as an absolute path. Unpack gzipped tar files.
gen_passwd	Use current user list and root password to generate new /etc/passwd and /etc/shadow files. Install new PAM configuration files. Keep a backup of original configuration.
restore_passwd	Restore original /etc/passwd, /etc/shadow, and PAM files.
warn	Run **lexis_warning** for the given number of seconds.
kill	Kill all processes with uid > 100. Unmount any network filesystems.
start	Write out firewall configuration. Write out server and port settings. Change to runlevel 4. Exit.
dump	Return a gzipped tar file of `/exam`.
ok	Return "ok".
quit	Restore original configuration. Clear */exam*. Change to runlevel 5. Exit.

Figure 3: lexis_active commands.

number of standard tools (xemacs for example) use writable directories for storing site packages, or similarly update-able files. While individual cases (like xemacs) can be fixed on a site-wide basis, it would be incredibly risky to include code to remove or hide such directories automatically.

For the time being, we make the assumption that /tmp and /var/tmp are the only world-writable local directories. If Lexis starts being used at a large number of sites, then more advanced techniques may become necessary.

The current lexis_startup is implemented in about 100 lines of perl.

lexis_warning

To warn any existing users that an exam is about to start we use lexis_warning, which is a simple Perl-Tk script that connects to the local X server. It turns the root window to a given colour (red by default) and pops up a small window containing a warning about the impending exam. The popup beeps in an irritating fashion every second until the current user acknowledges it. It is mainly intended for use when a Lexis session is scheduled during a normal lab period – it's not necessary when the rooms have been cleared and checked for a full official examination.

lexis_server

The Lexis server process is the heart of the Lexis system. It deals with data from a number of sources: there is a main config file, a network port for listening for new Lexis clients, the connections to Lexis clients, and also interactive input from the operator. The system is designed to enable one operator to manage many Lexis clients at the same time from the same server process.

The server is configured using XML. The DTD is shown in Figure 5, and Figure 6 shows an example config file.

The main config tag contains attributes describing where to store dump files and how often they should be taken, which port to listen on for booting Lexis clients.

While the config file defines "start" and "stop" times, they are for information only, as Lexis does not yet start and end exams automatically. It is technically feasible to trigger these events, but development time was tight, and the staff in charge of the exam were more comfortable retaining control over the start and end time of the exam in case of special circumstances.

Implementing auto start and finish would entail putting more critical code onto the client, which is something we wanted to avoid while the system develops. Also, the overhead of 200 machines all trying to dump to the server at precisely the same moment could cause problems on the server, and we didn't want to risk losing any candidate's work.

The rest of the config file contains a list of files to transfer to the clients, the list of candidate names, and a description of the hostnames of the client machines. The clients machines can be specified individually by name, or using a shortcut for ranges of machines. The example file would add the following machines as clients: lab25, dynamic01, dynamic02, ..., dynamic28.

Multiple server processes can communicate with the same client machine; each connection will spawn its own lexis_active process. We have used this technique with a modified lexis_server to create a separate dumping process in case of any problems or long-running jobs on the main server process.

```
<!-- DTD for LEXIS server config file -->

<!ELEMENT config (server*,file*,users,(machine|machine-range)+)>
<!ATTLIST config dump-dir CDATA #REQUIRED>
<!ATTLIST config dump-interval CDATA #REQUIRED>
<!ATTLIST config port CDATA #REQUIRED>
<!ATTLIST config rootpw CDATA #REQUIRED>
<!ATTLIST config start CDATA #REQUIRED>
<!ATTLIST config stop CDATA #REQUIRED>
<!ATTLIST config debug (0|1) "0">

<!ELEMENT server EMPTY>
<!ATTLIST server address CDATA #REQUIRED>

<!ELEMENT file EMPTY>
<!ATTLIST file name CDATA #REQUIRED>

<!ELEMENT users (#PCDATA)>

<!ELEMENT machine EMPTY>
<!ATTLIST machine name CDATA #REQUIRED>

<!ELEMENT machine-range EMPTY>
<!ATTLIST machine-range base CDATA #REQUIRED>
<!ATTLIST machine-range first CDATA #REQUIRED>
<!ATTLIST machine-range last CDATA #REQUIRED>
```

Figure 5: DTD for lexis_server config file.

Required perl modules: Term::ReadKey, FileHandle, File::Copy, DirHandle, MIME::Base64, MD5, IPC::Open2, IO::Socket, IO::Select, Net::DNS, XML::Simple

lexis_who

In order to find out which candidates had logged into which machines, we developed lexis_who, which is a simple perl script that queries the dump files stored on the server. It uses the files created on login to determine the user of the machine, for example .xsession-errors.

lexis_extract

Once the exam was over, we needed a way to extract specific files from the dumps, so that the answers to the various questions could be sent to the right marker. We wrote lexis_extract to achieve this, and to provide a framework for any other processing Lexis users might want to perform on the dumps. There are perl and ruby [12] versions of lexis_extract, with different default tasks. The ruby version is much more powerful than the perl version, and at 120 lines is twice as long.

Installation and Minimum Requirements

The minimum requirements for the Lexis client code are OpenSSH 2, Perl (with MD5 and MIME::Base64 modules), ipchains, and SysV style init. The processing requirements on the client are minimal; Lexis is designed to keep out of the way of the candidate as much as possible, so the greatest load on the system is likely to be any compilers the candidate is using. The Lexis client code is written in Perl, so it is possible for sites to customize the code to their specific requirements. Likewise, if other Operating Systems provide firewall rules in a similar way to ipchains, then Lexis can be ported to that OS (especially other UNIX variants). Lexis is not designed for Windows systems.

The Lexis client install consists of lexis_active and lexis_warning in /usr/local/bin, lexis_startup installed as a runlevel 4 startup script (and all other services removed from runlevel 4), a 'lexis' system group for ownership of /exam, and finally all clients will need the SSH2 public key the server will be using to contact them.

The use of lexis_warning is optional, and can either be omitted, or replaced with a suitable equivalent for the site in question. If you choose to use lexis_warning, the perl Tk module will also be needed. The Lexis client code can be easily made into an RPM or other package format. In which case, some additional security can be obtained by changing lexis_server to run

```
rpm -V lexis-clien && \
     /usr/local/bin/lexis_active
```

on the remote client machine.

The requirements for the Lexis server are somewhat stricter. The current lexis_server maintains a constant ssh connection for every client machine, there is also the overhead of MD5 and base64 on all client dumps, along with any processing of the dump files that needs to be done during the exam. We used an Athlon 800 with 512 MB of RAM to manage an exam with 160 client machines, but the machine was running very low on resources (we had to increase the file-max limit several times at the start of the exam to enable all the connections to succeed).

The main limitation is one of time – the server was originally written as a single thread, so as the

```
<?xml version="1.0" ?>
<!DOCTYPE config SYSTEM "lexis.dtd">

<config
  dump-dir="/var/lexis/"
  dump-interval="1 minute"
  port="334"
  rootpw="testpw"
  start="15/3/2001 12:00"
  stop="15/3/2001 13:00"
  debug="1"
  >

  <file name=".cshrc" />
  <file name="data_structures.c" />
  <file name="logic.pl" />
  <file name="skeleton.tgz" />

  <users>
mw foo bar
CAND001 CAND002 CAND003
</users>

<machine-range base="dynamic" first="01" last="28" />
<machine name="lab25" />
</config>
```

Figure 6: Config file for lexis_server.

number of client machines increases the time to complete each stage of the exam process rises significantly. With 120 client machines, every second that a client takes to complete a task equates to two minutes for the lab as a whole. 30 seconds is not an unreasonable time for a client machine to transfer all the files it needs, generate MD5 encrypted passwords for 100 users, shut down all non-essential system processes, change runlevel, and restart X. Unfortunately that means it would take an hour for the whole lab to startup. The current version of the server has some very simplistic multi-threading capabilities (call fork() for groups of 5 client requests), but it can still take a while for the whole set of client machines to complete intensive tasks. The initial startup is far and away the longest Lexis process; dumps and file transfers complete in a matter of seconds for the whole lab.

Security

First and foremost, Lexis is a security product. Its sole function is to provide a safe environment for taking exams. Its success is measured by how successful it is in that area: i.e., how secure is Lexis?

Client

If a candidate obtained root privileges, they would be able to circumvent or disable all the restrictions enforced by Lexis. For example, they would be able to drop firewall rules, connect to other hosts on the network, and access stored files via NFS.

Root privileges could be gained by a number of means: using the root password, rebooting the machine to single user mode, using a boot floppy, or installing a Trojan horse on the client machine before the exam. Lexis takes a number of approaches to prevent successful exploitation of any of these techniques.

The root password is unique to each Lexis exam, and is only stored on the local machine in an MD5 encrypted form. Any rebooting of the machine will generate a warning on the server when the ssh connection is dropped. The local LILO configuration is protected with a password to prevent booting in single user mode. The boot sequence can be re-ordered in the PC BIOS to prevent booting from floppy (although this cannot be easily automated).

Making use of a Trojan horse would require root access prior to exam, although even if this were done, set-uid binaries would not be effective. The greatest risk from an approach such as this would be to hide unauthorized information on the machine. The candidate would have to do this to all machines that might be used for the test in order for it to work. A tool such as tripwire [16] might be useful for checking system integrity if this sort of exploit were a concern.

In general, a large effort is required to subvert Lexis; easy attacks are already blocked, risky attacks such as rebooting would be easily visible to exam invigilators or the Lexis administrator during an exam, and other attacks require previous root access to the workstation, which could also be detected.

Server

The security of the server is of paramount concern; the root user on the Lexis server can get root access to any Lexis client. They would also have full access to the dumps. Lexis does not provide specific security for the server, as the setup will vary greatly depending on available tools, site policy, security awareness of academics involved in the exam, and also the general setup of the network (DNS servers, NFS servers if needed, and so on).

Lexis depends on DNS resolution for the forward and reverse lookup of client hostnames. This could be provided on the server, and so the server could then be firewalled exclusively to the Lexis clients. The approach we took was to use ipchains to restrict the server to the local network (not just the Lexis clients), and close all ports except ssh, while restricting ssh access to the minimum subset of users who needed access to the server for the exam.

The possible attacks we have considered are: security compromise by client, Denial of Service by client to prevent other candidates finishing exam, DOS from outside, security compromise from outside to tamper with stored dumps. None of these are easily solved by a simple toolkit approach – each Lexis server will have different security requirements depending on the importance of the exam, the environment, other uses of the machine, means of transferring submitted exam answers to markers.

The server is a much more traditional security problem than the Lexis client, as it needs to be secure before, during, and after exam. There is the usual compromise between ease and speed of use against security risks. The policy on each site must be the responsibility of the examiner, but a good basis is minimal services, firewalled to Lexis clients only during exam, encrypted dumps, restricted logins to exam personnel only. Lexis does not yet support encrypted dumps, but the feature would be simple to add, whether a symmetric key is set by the exam coordinator at the start of the session, or alternatively encrypting each dump for the users who are going to mark it (this depends on a reliable Public Key Infrastructure).

Lexis in Use

Lexis was developed in order to satisfy a requirement from the Department's Academic Committee that the First Year (freshman) undergraduate programming exam would be taken on lab machines. That requirement gave us a strict deadline for completion of development and testing of Lexis. The system would only be used if the examiners had been satisfied, through a demonstration, that it fulfilled their requirements.

While we were confident that the techniques used by Lexis were secure and met the needs of the examiners, we had no way of knowing how well the system would scale, how it would perform under load, and how it would cope with unexpected failures.

Early testing revealed a number of problems with the communication between server and client. Client crashes would cause a fatal error on the server when it tried to read from the filehandle connected to the client. Server crashes would leave zombie lexis_active processes running on the clients. These problems were successfully resolved by simplifying the client code and extending the server. We made the client block on input, so when the channel died, it would simply exit. The server was made much more resilient, trapping the PIPE signal, and removing clients from the active connection list at the first problem.

Unfortunately, these changes meant we had to sacrifice some functionality on the client; we had hoped to be able to asynchronously notify the server on significant events (login, logout, reboot, attempted network access, syslog messages), but there was no way to achieve this with the simpler client.

First Test

The first proper test of Lexis was supposed to be a normal programming test, much like the many that had been taken before, only this time with Lexis providing security. Unfortunately, a known bug in the lab software occurred during a demonstration of Lexis to the test coordinator. Even though the problem was completely unrelated to Lexis, the coordinator didn't feel confident enough to run the actual test with Lexis. There was a great deal of disappointment all round, and there was still the problem of successfully demonstrating a full Lexis test before the main exam two weeks later.

The day before the main exam, a number of students were due to sit another programming test. This would be the final chance for Lexis to prove itself before the big exam, and was run with the largest number of clients tried so far.

At this stage, the server was still using a single thread of execution, processing each client sequentially. It was painfully slow, but it was also reliable, coping with all the failure cases the Teaching Associates could think up – rebooting the client, unplugging a client completely and asking for the files to be restored elsewhere, deleting files and asking for the originals to be restored. Likewise, the system proved resilient against the security attacks they attempted – all unauthorized network packets were blocked. They tried sending mail, and although the command succeeded, the messages were only queued on the client machine, and could not be sent on until the firewall rules were lifted.

The test coordinator emailed us to say:

"Thanks very much. Lets hope it goes as smoothly tomorrow as it did today."

However, the speed issue was critical. With about 40 machines taking part in the programming test, it had taken over 30 minutes to get them all into an exam state. With 160 machines scheduled for use the following day, we could not afford a two hour wait for the system to start. Given that the system was basically reliable, and a complete rewrite was out of the question given the time restrictions, we needed to find a simple way to speed up Lexis operations.

The solution we settled on at the time was to use a very simple fork()-based approach: each request going to more than one client machine would be broken down into batches of five (selectable at runtime) and a new process forked to execute each batch. While this would increase our resource requirements, it increased the responsiveness of the system by an order of magnitude without compromising the security or reliability of the already-tested code.

First Lexis Exam: 21st March 2001

The computer labs were cleared the night before the main exam, and we started Lexis before the students arrived. While we had considered having a separate server for each area of the labs (this exam used four of the five rooms we had available), in the end we were able to coordinate and run the entire exam from one server. There were 160 machines, and 110 candidates.

The exam got underway with very few problems. One student had difficulty accessing files immediately after logging in, but transferred to a spare machine straight away. The problem turned out to be a corrupted filesystem from a prior hardware fault that noone had bothered reporting.

A short time later we received a number of reports of exam files being corrupted. Specifically, a library file provided by Lexis in /exam and vital to the exam was being over-written with binary data. This caused a minor panic among the exam administrators who had a number of distressed candidates unable to continue their work. It was very simple to send out fresh copies of the file in question to all the affected clients. That enabled the candidates to continue while we analyzed the cause.

Again, the problem was not actually caused by Lexis. An urgent investigation revealed that the library was being overwritten by graphics data, specifically a screen shot of the file manager. It turned out that one of the common keystroke combinations in the editor used by the candidates caused the file manager to dump a screen shot of the current window into the selected file. Once that was sorted out, the exam continued in a routine fashion.

We used lexis_who to print out a list of which candidates were using which machine, which was then checked off against the list prepared by the examiner. This revealed several machines where earlier errors had caused the server to drop the connection to the

client. We had assumed this would make the machine unusable for the client, but Lexis clients proved to be more robust than we thought, and the candidates were still using the machines. We added them back into the client list and they responded and started dumping again.

In response to this problem, lexis_server has been amended to check for dropped clients that should be active.

Automatic dumps were happening every five minutes for over three hours. In total we took 6600 dumps, totalling over 60 MB of data.

We received no complaints from the students, and those we spoke to after the exam were greatly in favour of Lexis exams over paper-based exams, especially for programming.

Conclusions

According to the BBC, on the 2nd of April 2001, students sat the first paperless exam in the UK in a pilot scheme in Northern Ireland [1]. In fact, we beat them to it by several weeks. Our system, Lexis, was used to administer a first year programming exam on 21st March 2001 which comprised 110 students with access to 160 Linux workstations and lasted for three hours. At the end of which, the labs were restored to general access use.

We believe that Lexis is the first general tool for managing on-line paperless exams on the Linux platform. Lexis enables computing skills to be securely examined in an environment that provides the same tools that the candidates are used to. Lexis can be used for any type of exam, from a multiple choice quiz to a full essay paper, although it is especially suited to situations where computers are a normal tool for the task in question.

Lexis is not designed to completely automate the process of University examinations – it won't start and stop exams by itself, won't grant extra time for latecomers, it can't mark the answers, and it certainly can't write the questions. What it can do is provide a secure framework for managing minimum privilege access to a local network of Linux workstations, while automatically backing up files at regular intervals. These facilities can be put to a number of uses, not limited to exams or tests.

One application that has been discussed with us is that of kiosk systems: a series of Linux workstations available for public use in an insecure environment. Lexis could be used to restrict user activity on the kiosk machines, while also restricting network access to securely maintained proxy servers for access to email or the web. This approach would significantly cut down the potential for abuse of the systems. The big advantage Lexis has over other approaches is that it works with very little modification to a standard installation. It doesn't require kernel patches or reboots.

Lexis was developed by system administration personnel to support an academic decision. The academics wanted a computer-based examination system for reasons of convenience, progress, and to satisfy student requests. The project progressed with the academics requesting features and suggesting failure scenarios, and the systems group suggesting pros and cons of various strategies and providing a system security perspective. Unusually for this type of collaboration, the academics were happy to accept the security restrictions, and the developers were able to provide all the requested features.

What does the future hold for Lexis? We have just completed another programming test with Lexis, and the coming academic year promises many more. We have also ported Lexis from our old RedHat setup to a new standard SuSE install. It took just one day to adapt Lexis to support SuSE-specific tools and configuration – the same code now runs on both platforms. Other Universities in the UK have expressed an interest in Lexis, and we would like to see it in use at other sites.

Availability

Lexis is released under the GNU Public License, and can be downloaded from http://www.doc.ic.ac.uk/~mw/lexis/.

Acknowledgments

Throughout the development of Lexis, the encouragement and support from other members of the department has been fantastic. Special thanks to Duncan White for help administering the main exam, Peter Cutler for his patience and understanding during testing, and all the students and staff who have taken part in Lexis exams.

The insight and guidance provided by Ozan Yigit have been invaluable while writing this paper, and we thank him for the care and interest he has shown in this project.

The Authors

Mike Wyer is a recent graduate of Imperial College who currently works as a Systems Administrator in the Department of Computing. He has had a long-term interest in examinations and computers, having worked on exam registration in a final-year group project. Reach him electronically at mw@doc.ic.ac.uk.

Susan Eisenbach is a Reader in the Department of Computing where she is Director of Studies, responsible for the teaching programme. Her research interests include programming languages for distributed computing.

References

[1] BBC News, news.bbc.co.uk/hi/english/education/newsid_1258000/1258446.stm.

[2] Blackboard, http://www.blackboard.com.

[3] Braun, Crable, "Administering Exams Electronically: Issues, Techniques, and Assessment," http://www.isworld.org/ais.ac.98/proceedings/track26/braun.pdf.
[4] Computing Support Group web pages, http://www.doc.ic.ac.uk/csg/.
[5] "init(8), Standard Sys V Root Process, ftp://sunsite.unc.edu/pub/Linux/system/daemons/init/sysvinit-2.78.tar.gz.
[6] Linux IP Firewalling Chains HOWTO, http://netfilter.filewatcher.org/ipchains/.
[7] MIT Kerberos, http://www.mit.edu/kerberos/.
[8] Linux, Linus Torvalds, http://www.linux.org.
[9] Linux Standard Base, http://www.linuxbase.org/spec/gLSB/gLSB/runlevels.html.
[10] Larry Wall, et al., Perl, http://www.perl.com.
[11] RedHat Software, http://www.redhat.com.
[12] Ruby, http://www.ruby-lang.org.
[13] Campen, San Diego State University, http://coe.sdsu.edu/eet/Articles/Paperless/start.htm.
[14] OpenSSH, http://www.openssh.com.
[15] SuSE, http://www.suse.com.
[16] TripWire, http://sourceforge.net/projects/tripwire.
[17] WebCT, http//www.webct.com.
[18] "What are MD2, MD4, and MD5?" http://www.rsasecurity.com/rsalabs/faq/3-6-6.html.

Dynamic Sublists: Scaling Unmoderated Mailing Lists

Ellen Spertus – Mills College
Robin Jeffries – Sun Microsystems, Inc.
Kiem Sie – Mills College

ABSTRACT

Unmoderated electronic mailing lists suffer from the paradox that the more successful they are, the more difficult they are to use and administer because of the increasing number of users and discussions. Traditional solutions to this problem, such as creating static sublists or providing a Web interface, are not always desirable. We discuss the problems with traditional solutions and present dynamic sublists, an approach to scaling online communities that increases communication opportunities without overwhelming users or administrators. We have built a system, Javamlm, implementing dynamic sublists, and report on its implementation, use, and future.

Motivation

Systers [2] is an unmoderated email list for women in computer science, originally created in 1987 for 12 women. The list has been tremendously successful at creating a community for people who had previously felt isolated and has grown to include over 2300 members in 38 countries. As Systers has grown, however, it has lost hundreds of members, especially senior women, because of the increased message volume that has come with increased membership. Additionally, members have felt less free to post to the entire list, instead responding directly to the sender of a message or not at all, reducing the sense of community and the utility of the list. Our goal is to provide a better format for such a large community than an unmoderated email list, while keeping the features that make Systers successful.

Rejected Alternatives

A common general solution to heavy volume on mailing lists is moderation, which can be performed top-down by a small set of administrators or bottom-up through collaborative filtering. While the most obvious cost of moderation is the human overhead required, a more fundamental problem is that moderation schemes limit diversity through the "tyranny of the majority." A topic of little interest to the majority may be of great interest and value to a minority. A better scheme (which we describe below) would allow users to customize which messages they see.

A conventional alternative to moderation is the creation of static sublists interested in particular subtopics. For Systers, there could be technical groups focused on Web design, groups discussing how to best deal with maternity leave, or support groups for pre-tenure academics. However, while an individual Syster may not be interested enough in maternity leave to join a group that discusses that exclusively, part of what members find valuable is hearing about other women's issues in a wide range of areas (especially ones that might affect her in the future). Thus, completely eliminating the sharing of information about maternity leave with the broader group lessens the value of the list. In the limit, there would be no common discussion among the entire group, just a vast collection of special interests. For these reasons, we rejected the idea of relying primarily on static sublists.

The most obvious way to support customized views is to go to a Web-based, newsgroup-like format, where Systers can browse the topics that interest them. Because it is not email-based, this solution would be unacceptable to many long-time Systers, who have expressed a preference for email, as well as to members in poorer countries with less Internet connectivity. It also removes the immediacy of the information, since most Systers would check the website at most daily or weekly, which would change the feel of the community.

Our Approach

We have a solution that we believe meets many of our design goals: dynamic sublists, which are similar to Usenet threads. Members are able to easily create, subscribe to, or unsubscribe from dynamic sublists. The opening message of a dynamic sublist is sent to all members. When a member signs up for Systers, she specifies whether she wants to see all messages or only the first message of each thread (a less precise but more user-friendly synonym for "dynamic sublist"). The default behavior is for users to be subscribed to all threads in order to mimic the behavior of the current system. Users may also specify whether they prefer to receive messages as plain text or as html.

Threads could exist for a limited period of time or be permanent. Under the current system, a volunteer collects job listings and posts them to the list once per week. She also forwards them immediately to

Systers who are looking for employment. Having a *systers-jobs* thread would eliminate the human administration costs while increasing functionality. An individual could specify any of the following behaviors:

- Receiving each job posting immediately
- Receiving digests of job listings on a daily, weekly, or monthly basis
- Never seeing job listings

At any time, she could change her preference without going through a human administrator.

Implementation

We have built a prototype, the Java Mailing List Manager (Javamlm), which implements dynamic sublists. Figure 1 shows the structure of the system. Information about users, threads, and messages is kept in a relational database. We are currently using Postgres as our database management system.

Our mail transport agent (MTA) is qmail, which has support for user-controlled mailing lists. Specifically, when incoming mail contains a hyphen in the local portion of the address (e.g., systers-subscribe@javamlm), the string before the hyphen is interpreted as the username, and the string after the hyphen is handled as specified by the user. In the case of Javamlm, qmail the message is piped to a Java program, which handles it as appropriate.

We have also built a Web interface to the database for users and administrators using AOLserver and Tcl. All of these tools are open source, which will allow us to release the entire package as open source.

Schema

The database schema is shown in Figure 2. The Subscriber relation contains information about each individual subscriber, such as name, email address, preferred message format (plain text or html), and whether or not to be subscribed to new threads. The Thread relation contains information about each thread, including a unique name and a reference to the initial message, which is represented in the Message relation, which stores meta-data about each message.

Because the Thread and Message relations reference the Subscriber relation, we cannot delete a record from Subscriber if a user unsubscribes. Instead, Subscriber has a boolean field, deleted, which is set to true if a user unsubscribes, inhibiting message delivery.

When a new thread is begun, the initial message is sent to all current subscribers. The SQL query is:

```
SELECT text_format, email
FROM Subscriber
WHERE deleted = FALSE
```

Subsequent messages within a thread are only sent to members who have expressed a desire to see them. A user's default preference for new threads is represented in Subscriber.preference. A value of 0 means "not subscribed to new threads," while 1 means "subscribed to new threads."[1] To honor a user's default preference, the SQL query for subsequent messages within a thread would be:

```
SELECT text_format, email, preference
FROM Subscriber
WHERE deleted = FALSE
AND preference = 1
```

The Override relation is used when a subscriber wants to override his or her ordinary preference for a specific thread. For example, if subscriber 99 ordinarily does not receive subsequent messages in threads (Subscriber.preference = 0), the following entry in the Override relation would indicate that she wishes to receive all messages in thread 157:

subscriber_id	thread_id	preference
99	157	1

Similarly, an Override.preference value of 0 indicates

[1]We chose to use an integer, rather than a boolean, representation, for expandability.

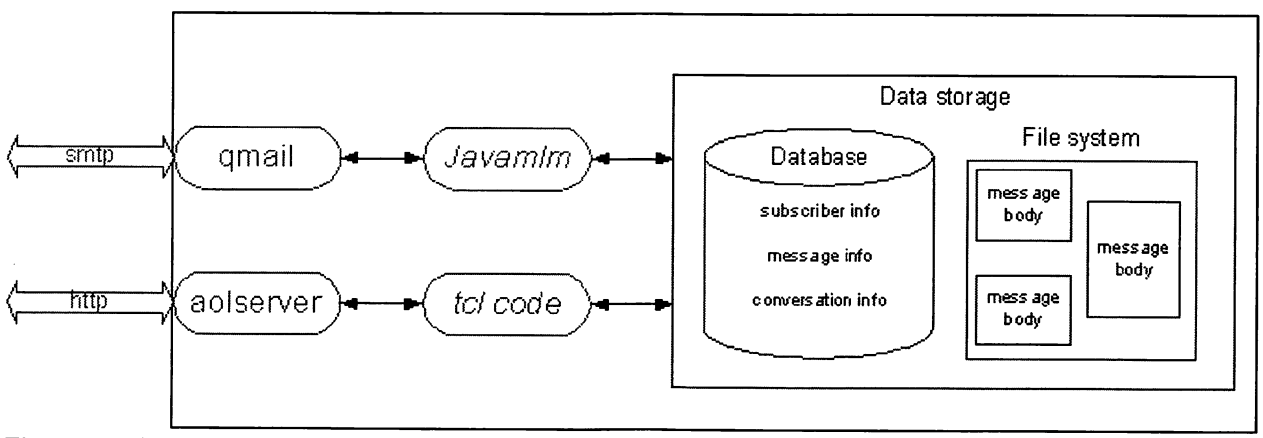


Figure 1: The structure of the system. Protocols are shown in arrows, processes in ovals, and store data in the box on the right. AOLserver and qmail are preexisting software packages. The authors' contributions are Javamlm, which provides mail-based access to the system and the tcl code, which is used for http-based access.

that a subscriber does not want to see further messages in a thread.

This is the complete SQL query to determine to whom a subsequent message in thread 37, for example, should be sent:

```
SELECT text_format, email
FROM Subscriber
WHERE deleted = FALSE AND
((Subscriber.preference = 1 AND
  NOT EXISTS
   (SELECT * FROM Override
    WHERE Override.subscriber_id =
    Subscriber.subscriber_id
    AND Override.thread_id = 37
    AND Override.preference = 0))
 OR
  (Subscriber.preference = 0 AND
   EXISTS
    (SELECT * FROM Override
     WHERE Override.subscriber_id =
     Subscriber.subscriber_id
     AND Override.thread_id = 37
     AND Override.preference = 1))
)
```

Messages

The bodies of messages are stored as regular files rather than in the database for the following reasons:

- Messages' highly variable size makes database storage awkward or inefficient.
- Locked access to messages is not needed, because messages are written once and never modified.

Subscriber

field	type	notes	sample value
subscriber_id	INTEGER	primary key	10328
email	VARCHAR(255)		"borg@iwt.org"
firstname	VARCHAR(16)		"Anita"
lastname	VARCHAR(16)		"Borg"
preference	INT2	Subscribed to new threads?	0 (no) 1 (yes)
format	INT2	Format to receive messages	1 (ASCII) 2 (HTML) 3 (both)
password	CHAR(16)	encrypted	FS9_eA%6
deleted	BOOLEAN	Has user unsubscribed?	false

Thread

field	type	notes	sample value
thread_id	INTEGER	primary key	157
thread_name	CHAR(16)	unique	fellowships51
sender_id	INTEGER	Person.subscriber_id	10328
base_message_id	INTEGER	Message.message_id	12000
subject	VARCHAR(255)	Subject of initial message	"AAUW fellowships"
status	INT2	Status of thread	0 (new) 2 (closed) 1 (in progress) 3 (perpetual)
parent	INTEGER	Thread.thread_id, can be null	null

Message

field	type	notes	sample value
message_id	INTEGER	primary key	12000
sender_id	INTEGER	Person.subscriber_id	10328
thread_id	INTEGER	Thread.thread_id	157

Override

field	type	notes	sample value
subscriber_id	INTEGER	Person.subscriber_id	10328
thread_id	INTEGER	Thread.thread_id	aauw23
preference	INT2	Only used to override default subscriber preference 0 (subscribed) 1 (unsubscribed)	

Figure 2: Database schema. The Subscriber relation contains information about each individual subscriber, such as the email address and whether or not to be subscribed to new threads. The Thread relation contains information about a given thread. Meta-data about a message is stored in the Message relation; the actual body is stored separately in a file. The Override relation is used when a subscriber wants to override his or her ordinary preferences for a specific thread.

- The file system provides an easier interface than the database for separate programs to access archived messages.

Javamlm accepts messages formatted as plain text, html, or both (through MIME multipart). Every message is converted, if necessary, to each of these formats, so users can receive messages in the format they prefer. Html is converted to text by the Lynx browser, which provides a command line option "-dump" for this purpose. Conversion from plain text to html is done through a modified version of the Perl script otxt2html, written by Ola Lundqvist [11]. Once converted, the messages are stored to disk. For example, the body of message 120 to Systers would be stored in the files:

- ~systers/javamlm/archive/120.text-plain
- ~systers/javamlm/archive/120.text-html

MIME-compliant messages are assembled through calls to the javax.mail package. To accommodate all of the users' preferences, up to six different messages are created: in each of the three formats (text/plain, text/html, and multipart) and with two different footers (unsubscription and subscription information), which are described more in the next section.

The program iterates through each of the subscribers, choosing the correct message format, specifying the email address, and creating a variable envelope return path (VERP). VERPs, invented by Dan Bernstein, automate matching bounced emails with subscriber addresses [2]. For example, a message to ada@lovelace.com on the Systers list would have a VERP of systers-error-ada=lovelace.com@javamlm.mills.edu. This is placed in the SMTP "From" field [9], which instructs mail transport agents (MTAs) where to send error messages. No matter how many times the message is rerouted through ".forward" files and other mechanisms [15], if it eventually bounces, the subscriber information will be intact, allowing the subscriber to be retried or unsubscribed. Note that message headers [12] are not personalized for each subscriber, only SMTP headers [9], minimizing the per-subscriber overhead.

User Interface

Figure 3 shows the Web form for joining a list. Other forms allow users to change their membership options and allow the administrator to access and modify members' settings. Web access is password-protected, using cookies. Currently, messages cannot be sent or viewed through the Web.

Users' primary method of access is through email. A member of Systers creates a new thread by sending the introductory message to systers-new@javamlm.mills.edu. Alternately, if she wishes to name the thread "fellowships," for example, rather than having a system-assigned name, she would send her introductory message to systers-new-fellowships@javamlm.mills.edu. All members would then receive the first message in this new thread. Figure 4 shows a sample message as it appears to the sender and an html-enabled recipient. Note that a number has been appended to the thread name, "fellowships," to make it unique.

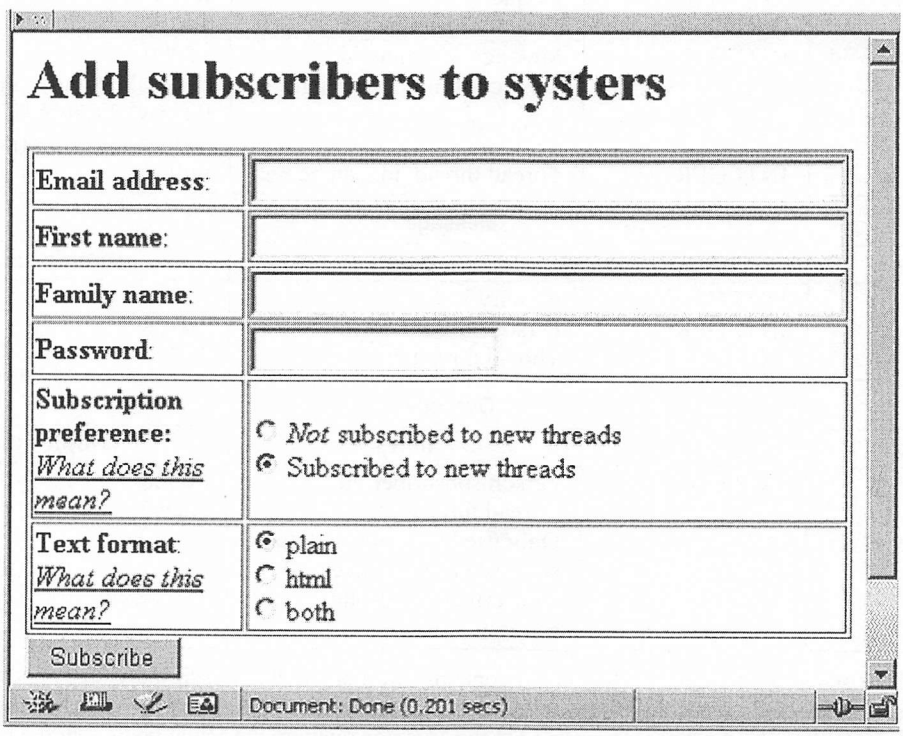


Figure 3: Sign-up screen.

If a recipient wishes to reply to the sender, she uses her MUA's[2] "reply" function. If she wishes to contribute to the thread, she uses the "reply-all" function.

Unless a recipient specifies otherwise, her default preference (e.g., "view all messages in new threads") would be in effect for subsequent messages in the new thread. There are two ways of overriding her default preference, by email or through the Web. As Figure 4 shows, instructions for unsubscribing from the thread appear at the bottom of the message to users who are by default subscribed. Similarly, if a user is by default not subscribed to new threads, instructions for subscribing are included.

Unsubscribing by email takes at least two steps: activating the embedded mailto link (e.g., mailto:systers-unsubscribe-fellowships51@javamlm.mills.edu) and confirming to the MUA that the email should be sent. (As RFC 2368 points out, it would be insecure for a MUA not to require user confirmation [7].) To minimize the number of required user actions, we also provide in each message a Web interface for overriding one's default preference, as shown in Figure 4 (e.g., http://javamlm.mills.edu/scripts/override?listname=systers&thread=157&preference=0). Note that the *subscriber_id* is not included in the URL, for three reasons:

1. It would be inefficient. Including a different *subscriber_id* in each message would require a unique Java MimeMessage object for each recipient.
2. It would be insecure. If the *subscriber_id* were embedded, it would be easy for a subscriber to unsubscribe other people by substituting their *subscriber_ids*.
3. It is not necessary. Because the Web interface uses cookies to store the *subscriber_id* of users who have successfully logged in, it is not necessary to reenter it. If there is not a valid cookie, the user is prompted for her username and password. Users who share computers can explicitly log out of the system.

[2]A mail user agent (MUA) is a client program for a user to access and compose email. Common MUAs include Eudora, Outlook, RMAIL, and pine.

Experience

We performed a month-long user test with 20-40 members of the main Systers list. Unfortunately, we were unable to generate a critical mass for sustained thread. Still, we were able to extract some useful information.

Of the 40 subscribers, 39 chose to receive all messages by default (i.e., preference = 1). Twenty-six members specified plain text as their preferred format, six hypertext, and eight both (i.e., MIME multipart). Nobody unsubscribed from the list during the trial period. We were surprised that the majority of users chose plain text over hypertext, underscoring the importance of having a good ASCII user interface.

Seven of the 11 threads were created by the system's authors. In an attempt to encourage members to use the unsubscribe feature, we created a thread, called *word1*, that forwarded the daily word from Anu Garg's A-Word-A-Day service (http://wordsmith.org/awad). Six members unsubscribed from the thread without difficulty. One member publicly responded positively to the thread with her thoughts on a word,

```
To: systers-new-fellowships@javamlm.mills.edu
Subject: AAUW Fellowships
From: spertus@mills.edu

Information about AAUW fellowships is now
available at http://www.aauw.org.
```

Figure 4(a): Initial message.

```
To: systers-fellowships51@javamlm.mills.edu
Subject: SYSTERS: AAUW Fellowships
From: borg@javamlm.mills.edu
Senders: systers-fellowships51@javamlm.mills.edu

Information about AAUW fellowships is now available at
http://www.aauw.org.

To unsubscribe from this thread, send email to
systers-fellowships51-unsubscribe@ javamlm.mills.edu or visit
http://javamlm.mills.edu/scripts/override?listname=systers&thread=157&preference=0.

To unsubscribe from systers, send email to
systers-unsubscribe@javamlm.mills.edu.
```

Figure 4(b): Message as viewed by recipient.

Figure 4: The first message in a new thread, as (a) written by the sender and (b) seen by the recipient. This assumes that the recipient has hypertext enabled and is by default subscribed to new threads. The recipient will continue to receive further messages in this thread (i.e., sent to systers-fellowships51@systers.org) unless she activates the mailto orhttp link to unsubscribe from the thread or from thelist.

which motivated one of the authors to create a new thread asking members for theirs pet peeve about word misuse. It was unclear how to begin this new topic. If it were within thread *word1*, subscribers could not avoid the discussion without unsubscribing from the A-Word-A-Day message, which many enjoyed. Instead, we created a brand-new thread, *peeves1*, which had the disadvantage of going to people who had unsubscribed from *word1*. This showed the desirability of allowing for threads to have children, which would allow child threads to begin by going only to subscribers of an existing thread instead of to all users.

The first message in the *peeves1* thread was created by sending mail to systers-new-peeves@javamlm, which was cc'd to the subscriber who had first expressed her opinion. Because this member received the message directly (and not just through Javamlm), the to-line contained systers-new-peeves@javamlm instead of systers-peeves1@javamlm. Thus, when she replied, another thread, peeves2, was created. We are exploring solutions to this problem.

A point that no users raised but that concerned the authors was how to deal with individuals' joining the list while a thread is in progress. If a user is by default unsubscribed to threads, she will not see any messages until a new thread begins (which may be the right behavior). We may want to provide new users with the options of being filled in on active threads, perhaps through a Web interface.

A related problem is what to do if a user unsubscribes from a thread (either by default or explicitly) and then explicitly subscribes. Currently, she would never get the messages that occurred while she was not subscribed. This is a particular problem for users who are unsubscribed from new threads by default, since, by the time they explicitly subscribe to a thread of interest, earlier messages may be lost to them. Clearly, we need to design a better approach.

Another problem that arose was varying hypertext production by different MUAs. The Eudora mailer encloses a formatted message with "html" tags, while Yahoo! Mail does not. Javamlm failed to properly insert (un)subscription information in messages formatted differently from expected.

One user requested that subscribers be emailed their user name and password, a feature provided by Mailman [14]. We had opted not to provide this feature both because of the insecurity of email and because we only store encrypted passwords. Clearly, some users prefer convenience to greater security, suggesting that we should provide per-user or per-list security options.

Related Work

While many people have addressed minimizing administration cost for owners of large mailing lists [e.g., 1, 4, 5, 14], our goal is more to minimize costs for *users* of high-volume mailing lists without increasing administrative overhead.

Our system is a direct descendant of the threaded news reader (trn) for Usenet [8] by Wayne Davison, which grouped messages with common ancestors into threads, allowing users to read articles by thread, instead of by date or subject, or to automatically avoid further messages in the thread. While the introduction of threads revolutionized newsreaders, it has been slow to cross over to email lists.

Ka-Ping Yee built a system, Roundup [16, 17, 18], supporting "fine-grained mailing lists," making use of the "In-Reply-To" header [12]. He used the term "issue" for what we call a "thread" or "dynamic sublist." Each issue has a corresponding set of users who receive new messages, called "nosy lists." Users are added to an issue's nosy list if they are found in the "From," "To," or "Cc" headers of a message within the issue.

Jamie Zawinski has implemented and documented [19] an algorithm for threading email messages based on the "In-Reply-To" [12] and "References" [8] headers. Mark Crispin and Kenneth Murchison recently described the algorithm more formally and proposed changes to the IMAP protocol for server-side support for grouping messages into threads [6]. While making the best of the current situation, this approach is limited by MUAs' and users' inconsistent use of the "In-Reply-To" and "References" headers. For example, there's nothing to stop a user from manually copying the "to" field in replying to a message, rather than using a MUA's reply-to function. By embedding thread information in the "to" address, our system forces MUAs and users to include it in replies. Combined with Crispin and Murchison's proposed IMAP extensions, more powerful server-side threading could be provided.

Future Work

We consider our current system, Javamlm, to be a prototype, which we do not plan to extend. Instead, we will incorporate support for dynamic sublists into Mailman, the GNU mailing list manager [14]. We chose Mailman because it has the following features:

- Web-based interfaces for both administrators and users.
- Per-list and per-user configurability, including digesting.
- Automatic web-based archiving.

We plan to extend each of these features, such as allowing users to specify that a specific dynamic sublist should be delivered in digest form, while others should be delivered immediately. This would enable the *systers-jobs* example described earlier, allowing job-seekers to receive announcements immediately and other users to receive announcements as weekly digests or not at all.

Unlike the *jobs* sublist, most dynamic sublists are expected to be active for a short period of time (days or weeks). Planned future functionality would

give users the additional option of seeing only a sublist's first message and a final summary message created by the thread's originator. We also plan to support individual keywords so a subscriber can automatically see all messages containing "linux" and "administration" if she so specifies.

We will also address the issues raised from our user test, such as the best means for creating child threads, supporting mid-thread subscription, and per-user security options. We welcome further input and participation in the project by anyone interested.

Acknowledgments

We are grateful to Sara Kiesler, Jennifer Goetz, and Lee Sproull for performing a study of Systers members with Robin Jeffries, upon which this work is based. We would also like to thank Anita Borg and the Systers members who participated in those surveys and informal discussions, offered many ideas for redesigning of Systers, and volunteered their skills and effort to improve the Systers community. We have also received useful input from Gloria Montano, who led the first user test. Our understanding of email protocols was greatly enhanced by members of the *List Managers Mailing List*. We received encouragement and valuable feedback, including pointers to related work of which we had been unaware, from the LISA referees, Sigmund Straumsnes, and Mailman author Barry Warsaw. Ellen Spertus and Kiem Sie are partially supported by a National Science Foundation Faculty Early Career Development grant. This work is being done in cooperation with the Institute for Women and Technology, which hosts Systers.

References

[1] Bernstein, D. J., "Ezmlm," http://cr.yp.to/ezmlm.html.

[2] Bernstein, D. J., "Variable Envelope Return Paths," http://cr.yp.to/proto/verp.txt, February, 1997.

[3] Borg, Anita, "Why Systers?" *Computing Research News*, http://www.systers.org/keeper/whysys.html, 1993.

[4] Chalup, Strata Rose, Christine Hogan, Greg Kulosa, Bryan McDonald, and Bryan Stansell, "Drinking from the Fire(walls) Hose: Another Approach to Very Large Mailing Lists, LISA XII," 1998.

[5] Chapman, D. Brent, "How I Manage 17 Mailing Lists Without Answering '-request' Mail," *LISA VI*, 1992.

[6] Crispin, Mark R., and Kenneth Murchison, *Internet Message Access Protocol – Thread Extension, Internet Draft*, IMAP Extensions Working Group, IETF, 2001.

[7] Hofman, P. L. Masinter, and J. Zawinski, *The mailto URL scheme (RFC 2368)*, Network Working Group, IETF, July, 1998.

[8] Horton, A. and R. Adams, *Standard for Interchange of USENET Messages (RFC 1036)*, Network Working Group, IETF, December, 1987.

[9] Klensin, J., ed., *Simple Mail Transfer Protocol (RFC 2821)*, IETF, April, 2001.

[10] Lindberg, Fred, "Ezmlm/idx Manual," version 0.32, http://www.ezmlm.org/ezman-0.32/index.html, March, 1999.

[11] Lundqvist, Ola, "otxt2html.pl," http://www.opal.dhs.org/programs/otxt2html/index.oml.

[12] Resnick, P., ed. *Internet Message Format (RFC 2822)*, Network Working Group, IETF, April, 2001.

[13] Rosenthal, Chip, " 'Reply-To' Munging Considered Harmful," http://www.unicom.com/pw/reply-to-harmful.html, May, 1999.

[14] Viega, John, Barry Warsaw, and Ken Manheimer, "Mailman: The GNU Mailing List Manager," *LISA XII*, 1998.

[15] Westine, A. and J. Postel, *Problems with the Maintenance of Large Mailing Lists (RFC 1211)*, IETF, March, 1991.

[16] Yee, Ka-Ping, "Roundup: A Simple and Effective Issue Tracker in Python," short talk, Eighth International Python Conference, 2000.

[17] Yee, Ka-Ping, "Roundup: An Issue Tracking System for Knowledge Workers," design proposal, Software Carpentry Design Competition first round, March, 2000.

[18] Yee, Ka-Ping, "Roundup: An Issue Tracking System for Knowledge Workers, Implementation Guide," Software Carpentry Design Competition second round, June, 2000.

[19] Zawinski, Jamie, "Message Threading," http://www.jwz.org/doc/threading.html, 2000.

GEORDI: A Handheld Tool For Remote System Administration

Stephen Okay – Road Knight Mobility Labs
Gale Pedowitz – Protura Consulting, Inc.

ABSTRACT

This paper discusses the design and implementation of a tool for allowing technical staff to perform diagnosis, triage and remediation of system problems from a commodity handheld device (e.g., a PalmOS PDA) with a wireless network connection using industry standard encryption and privilege management software. We argue that this model is equivalent to using a desktop or laptop from a security aspect but is more convenient and efficient given its minimal resource requirements, "instant-on" availability and usability from arbitrary locations.

We explore previous work in this area and posit that our solution offers significant advantages because it adopts the stylus and forms-based usage model prevalent on many PDAs rather than trying to overlay the classic command-line interface onto a system which was not designed to use it.

Finally, consideration is given to how small mobile systems, like the one used in this tool, will impact the task of system management in the future in terms of benefits and risks.

Introduction

The life of an on-call system administrator (sysadmin) is an interrupt-driven endeavor. When not beset by users or machines demanding his or her instant attention, he or she is surrounded by an array of pagers, PDAs, cell phones, and other messaging systems, all eager to alert him or her to fresh disasters. These seem to go off most frequently when the sysadmin is within range of being alerted about a problem but a significant distance from the nearest point from which to affect a solution.

No longer limited to displaying just the same numeric string much of the time, many of these devices are sophisticated two-way, programmable messaging systems with considerable memory and CPU power. Sometimes they aren't separate devices, but software built into cell phones or PDAs. Despite all these enhancements, the response to them is often the same as it has always been: to walk, run or drive to the nearest terminal to fix the problem. This is not only irritating to on-duty staff, it's wasteful and expensive, especially when the problem is often something as simple as "the $DAEMON died" or "$SERVER needs to be rebooted." In the current climate where users and equipment are scattered across time zones around the globe, the answer "I'll be there as soon as I can" is not an acceptable response.

The General External Operators' Remediation and Diagnostic Interface(GEORDI) offers an alternative to the scenario mentioned above. GEORDI provides a method for the diagnosis, triage and remediation of system problems from the sysadmin's current location. This is accomplished via a commodity PDA and wireless TCP/IP connection. Using existing industry-standard software such as ssh [Ylonen] and sudo [Courtesan], GEORDI creates a secure connection to a remote host without requiring additional host software. This approach minimizes the likelihood of introducing new exploits simply through the use of this access method. Interaction with GEORDI is through the native handheld user interface rather than the more traditional console/keyboard paradigm.

We chose to focus our research in addressing this situation on the PalmOS family of PDAs. They hold a commanding market share over other handheld devices and offer mature development environments across multiple platforms such as Metrowerks' "Code Warrior" for Microsoft Windows and Apple MacIntosh platforms, and the prc-tools GCC toolchain and pilrc resource compiler for many UNIX systems. The general architecture presented should be easily adaptable to other platforms since the software it is based on is available under one or more open source licenses.

An Itch to Scratch

Like many other tools for system administrators, GEORDI came about as a result of the authors trying to scratch an itch. For one, we found that we were still traveling to sites or terminal rooms seeking out remote access for problems that were often trivial to solve in comparison to the effort expended to address such issues. Frequently, we were tasked with driving to work to restart a server or application whose death had been broadcast to our pagers. While emergency maintenance has always been a part of a system administrator's job, we found it increasingly frustrating in light of the growing sophistication of alert devices. Indeed, it was becoming commonplace for system professionals to carry CPU power exceeding that of recent

legacy desktops. While it would not be possible to obviate every instance of on-site repair, we theorized that many midnight treks to the server room could be prevented with effective application of the devices we always had with us.

The other facet of this was the desire to carry one's system environment with oneself and to be able to copy it to new locations. Not every situation would involve a fresh disaster; even during normal operations, it would be advantageous to take the current state of a task in progress and move it to another system or device. Desktop layouts, terminal window states, and even programs and scripts in execution should be portable across both large systems with high-speed connectivity, and smaller, less connected ones as well.

Finding significant areas of intersection between these two efforts, we decided to work together on a tool that could serve as a sort of remote "first aid kit." It would let a sysadmin collect data on system health and activity, allowing him or her to make a decision if a trip to the site was warranted or if an issue could be addressed from his or her current location. This would require research in the areas of UI design, connectivity and security as well as investigations into the systems architecture of small, mobile computing platforms.

The initial goals behind GEORDI then were:

- To provide a way to quickly examine and fix system problems from arbitrary urban locations. It would be nice to restart your webserver from the middle of the Gobi Desert, but that kind of scenario is far more dependent on industries and technologies not under our control. It is best to focus our energies on those things more within our grasp.
- To do so using common, commercially available handheld systems such as PDAs, palmtops, smart pagers, etc. With CPU speeds between 33-200 Mhz, 8-64 MB RAM and 16-bit color displays, we now carry on our persons what used to sit on our desks 5-10 years ago. Additionally, most IT departments are unlikely to fund or otherwise be able to justify the purchase of additional single-purpose gadgetry, especially for a large staff.
- To do so in such way that poses no greater security risks than already exist through current remote access methods.
- To do so within the realm of current open source software licenses so that others have the chance to build on our work to suit their own needs.
- To encourage research and debate on what future tools for performing system admininistration tasks should look like, with the focus on a practical, needs-based UI that's as portable as possible. Users and equipment are spreading around the globe and we're expected to keep up with them. This can't always be done from a 17 inch display with a 600 Mhz CPU connected to a T1 in the datacenter.

Applications and usage models

Desktops vs. Handhelds

The idea of remote access from handheld devices is not entirely novel. Indeed, the presence of terminal applications on small keyboard-based organizers can be seen as far back as the early 1990s in the HP95LX palmtop and Sharp Wizard/Zaurus. At the time of this writing, there was even a terminal client and Lynx web browser for the HP 48 calculator [Costar]. Handheld computers themselves go as far back as the early 1980s with the Sharp 2100N, marketed in the US as the Tandy PC-1 Pocket Computer.

Many connectivity options exist for more recent handhelds. Numerous VT100 [Hall], telnet [Ptelnet], ssh [GoldbergSSH] and other tty-like clients are available for most modern PDAs, including the PalmOS family of PDAs and PocketPC. The reasons for this proliferation of terminal clients is rather obvious. Now, as then, the world still largely runs on the command-line when it comes to systems and network management. Graphical status tools may be ideal for displaying status and performance; to actually affect or control a system remotely from an arbitrary location, however, the only reliable constant is a terminal.

While these applications present the sysadmin with the familiar command line interface, any attempt at sustained work will demonstrate that using these tools on a handheld is quite different from using them on a traditional system. The screen is smaller, and the keyboard is often laid out differently, if there is one. The API for the handheld may provide only limited support for text display outside of native forms-based support, resulting in emulation bugs or requiring termcap entries specific to each terminal application and handheld used on the remote host. The PalmOS API, for example, has no concept of an actual console and all characters must be draw on the screen using bitmap coordinates with one of the WinDrawChar() functions [PalmOS]. Scrolling, placement and update are likewise left as an exercise to the individual application.

One approach to dealing with this is the one taken by tools such as VNC [Richardson, et al.] that provide a complete pixel-by-pixel replication of the remote system's screen environment. VNC also provides state-preservation that allows a user to move mid-keystroke from, say, a Sparc desktop to a Windows PC or Mac and pick up typing exactly where he or she left off. Depending on feature support in the client, these events can even be reflected back to the remote system the user just left, providing a useful means of remote support in a heterogeneous environment.

The chief problem here is largely one of bandwidth. VNC was originally designed for use in an environment where the primary network media is

ATM and the effort involved in shipping around chunks of a 1024x768, 24-bit desktop image is comparatively small. This continues to work well over wires at 10/100 Mbit speeds but begins to suffer performance problems once the pipe drops below a T1. The situation worsens when one considers that the maximum speed for the PalmOS serial or IRDA port is 57,600 bps. In informal tests with PalmVNC on a 33 Mhz Visor Platinum, we found approximately a 5-8 second delay between the time one initiated an action on the Visor and the time that action was reflected back to us on the screen. When the connection is reduced to 19,200 bps, a speed reasonable to expect from CDPD or micro-cel carriers, the delay shoots up to 30-48 seconds. There are 802.11 modules available for a number of different PDAs on the market, but their short range limits their usefulness and can necessitate a significant infrastructure investment to provide coverage for a campus environment.

Numerous UNIX vendors and individuals at one time or another have made some effort in non-CLI-based administration interfaces, each meeting with varying degrees of success. We chose to revisit those tools we had encountered previously in our careers,such as the Solaris AdminTool, IRIX System Manager and AIX SMIT. While they were generally focused only on administration of the local host they ran on, each came pre-installed on new systems from their respective vendors by default and each had a different approach to the idea of system admin shells. The Solaris AdminTool focused primarily on the novice user, providing a way for him or her to quickly and easily perform user account management and peripheral/device control. SGI's System Manager extended this somewhat further by building the tool into the normal user desktop menubar and providing greater detail on the status and configuration of disk,network and other peripheral devices. SMIT differed significantly from the other administrative shells. Overall, it was probably the most directly useful to our work; its ability to record keystrokes as one stepped through its menu-driven UI and to convert these into shell scripts provided some of the inspiration for what later became GEORDIs Command Builder.

In each tool, we also encountered a number of deficiencies.

- Their proprietary nature and lack of support for managing systems remotely limited their usefulness, even within the vendors' product families.
- They tended to do one or two things well, but fell significantly short in other areas.
- Some tended to exhibit poor state/error control. Pushing the button and not getting an error dialog didn't always mean things worked. It was often necessary to drop to a shell to confirm the operation occurred as expected.

A subtle yet more crucial drawback in all of these tools was that they were designed for large, general purpose systems with a usage model that involves sitting down and committing to several minutes (or more) at console. It is almost expected that the user won't finish the task or find what he or she is looking for without some digging, so the system is designed to accommodate and, in some ways, encourage this.

Handheld systems in contrast follow a completely different usage model. They are used for seconds at a time as people pick up the phone or dash through the airport. They are tossed in the car and then glanced at furtively for directions while we drive down the road. If a handheld exhibits any sort of "boot time" or forces the user to stop and focus on the unit, rather than the data it contains, it has failed as a useful device. Consequently, the data these devices store is equally brief. IP addresses,dates, passwords, error codes, building numbers and other similar scraps of information fill their memory.

How this affects a system administrator's ability to efficiently do tasks on one of these devices is not immediately apparent until the first time he or she must scribble out something like

```
ps -aux | awk '/luser/ \
  {printf(''killall %s0', $9) }' | sh
```

on a PDA terminal client using the stylus instead of a real keyboard.

This is, as one might guess, far from optimal. What can be done? As system administration frequently requires the ability to interact with a system at its lowest levels, access to the command line is critical, even if it is awkward and inconvenient to use in some situations.

GEORDI Design

UI Design

Initial Efforts

The GEORDI UI began life as something similar to a desktop GUI, but with an emphasis on being highly configurable and extensible. Menu items could be moved around and re-ordered, so that the frequently selected choices could reside on top. Commands and scripts, represented graphically, could be linked together by dropping them on or next to each other. The primary focus here was on modularity and configurability. We felt that we already had one or two strikes against us from a industry cultural perspective simply by virtue of the graphical nature of our tool.

This initial version was passed around the table at local user group meetings, and the responses were largely negative. The UI was too confusing for the available screen real estate and quickly led to users being lost in a maze of icons, tear-off menus, drop-down lists and the like. The one tool they did seem to find and use consistently was the CLI popup, totally defeating the purpose of the tool. In the drive for

maximum configurability, we had inadvertently recreated those tools we resented so much on the desktop.

In a number of ways, this turned out to be a blessing in disguise. Writing a Palm app using almost nothing but the Gadget resource involved some rather heavy lifting in the code. As this was only the first revision,things were bound to become worse. We still felt we were on the right track with the "LEGO Brick" model as most of the complaints seemed to focus on screen size and performance issues.

Squeak

Squeak is a modern implementation of Smalltalk-80 [Squeak], one of the original object-oriented programming languages and the progenitor of much of the OO programming movement in the 1980s. An implementation is available for the Compaq iPAQ handheld. With a 200 Mhz StrongARM CPU, 32 MB of memory and a high-resolution color screen, the iPAQ is a considerably more capable system than the Palm. Squeak also boasts a much richer set of UI and data type primitives than PalmOS. Additionally, Squeak's VM architecture offers write-once portability, and the prospect of having usable Geordi clients for any of Squeak's many supported architectures was extremely attractive. A Squeak version of GEORDI similar to the aforementioned "Brick" architecture was fashioned in short order, and a feature was added where one could drop scripts as objects onto a small TTY in the corner of the screen. Unfortunately, this nascent effort was shelved just as quickly as the first Palm implementation; serious bugs in Squeak's graphical interface manager made testing and debugging impossible. These issues led to our decision to concentrate exclusively on the Palm implementation of Geordi for this phase of our work.[1]

A Forms-based UI

We noticed[2] that many of the UNIX commands used for reporting system status and configuration produce output in a row/column format. The PalmOS API, coincidentally, provides extensive support for row/column table forms with a callback handler mechanism allowing users to perform actions on the data in those forms.

By wrapping the output inside PalmOS form elements rather than just writing it out to the screen, we are able to attach event handlers to each table cell, allowing a variety of actions or additional related dialogs or forms to be displayed in response to each tap. Through this we are able to provide a coherent way to execute and display the results of commands such as top(1), df(1), netstat(8), ifconfig(8), etc. In retrospect, this seems quite an obvious approach, but we are not UI experts and there is a cultural bias in our profession that seems to say "If it doesn't have a console mode, it can't be powerful enough to be useful."

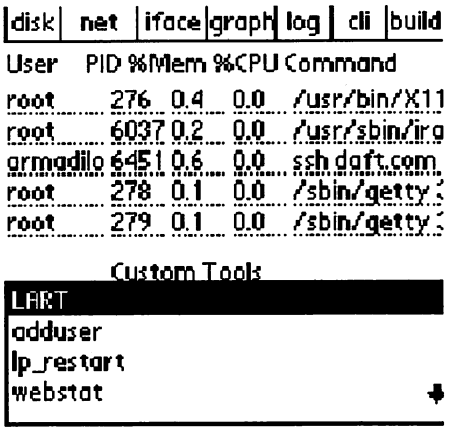


Figure 1: Process listing from remote host.

In Figure 1, we see a process listing from the remote host. The buttons across the top of the screen lead to similar forms, which provide access to other common system tools like those mentioned above. To send a signal to process 278, we tap on the PID column and are presented with the display in Figure 2.

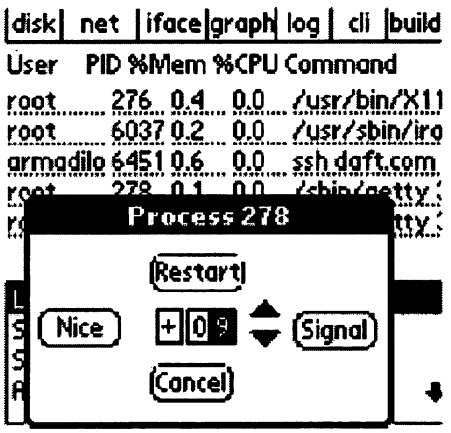


Figure 2: Signaling process 278.

From this dialog, we can easily set the nice value of the process or send it any valid UNIX signal. Similarly, to see what PID 278 is running, we can tap on the ps table column containing the command name to see the full text of the command being executed under that PID.

Finally, if necessary, we can pop up a text form similar to the one in Figure 3 to enter a specific command.

This experience taught us an important lesson in UI design and usability. Good tool design should flow from both a specific need and philosophy where the manifesto of the tool precedes its actual creation. The most successful UIs are those which impart a "Zen of ..." design on their applications. They fill an actual need, not a marketing goal, and are consistent in form

[1]This has since been addressed and resolved as of August 2001. Returning to this platform is one goal of future work

[2]Actually, it's more like we were knocked on the head with this by our friend and colleague Jim Dennis. After testing yet another version of the Palm GUI, Jim observed that he "just want(ed) a ps table where I can tap on things and kill them!"

and function. The success of the Mac and Palm are very much tied up in this. We feel GEORDI follows this because the forms and other UI elements it uses are the same as those found in the DateBook or PhoneBook. Even command-line tools share this to some extent. -V usually will give a version number, -v verbose output, -q quiet operation a single – to represent standard I/O and so on.

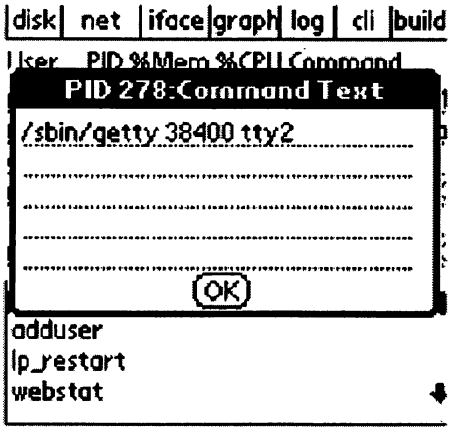


Figure 3: Text box for command.

Communications Issues

There are a number of communications issues that present potential problems for using handheld devices for remote system administration and monitoring. Service availability is a major one. As of this writing there are only a handful of wireless data carriers, and even fewer whose networks provide IP connectivity. Instead, most wireless carriers are based on store-and-forward or other non-persistent, non-stateful technologies. One example of this is the Palm.net service for the Palm VII family of wireless PDAs, which uses the BellSouth pager network. Under this "transactional" model, data requests are sent from the PalmVII to a proxy server which then converts them into proper HTTP requests to retrieve the web page requested by the Palm VII. A separate connection returns the requested data to the PDA.

For those networks that do provide a PPP or PPP-like service on their network, the picture is better, but not by much. Connection speeds range between 9600 and 38,400 bps. For our work, we used the OmniSky service, which is a CDPD network supporting connections up to 19,200 bps.[3] There can be significant delays on the order of 20-60 seconds, or more, to establish a connection to a remote site and then the same or longer during the lifetime of the connection. Often, latency on the wireless part of the connection can exceed the timeouts for a TCP/IP connection, causing it to drop and forcing the user to restart his or her session. This leads to the interesting result where a simple network like Palm.net can end up providing a better overall user experience. Since it does not employ protocols which depend on a continuous connection to function, it appears to the user to be more robust.

The Bandwidth Basement

While networks like Palm.net may not allow for protocols like ssh to be run over them, the bandwidth available is still sufficient for our purposes. The PalmOS Web Clipping Developer's Guide recommends that applications using the Palm.net service should send no more than 70 bytes of payload data per query and receive no more than 360 bytes per response [PalmPQA]. This is after a 5-60% compression ratio of the transmitted data, depending on whether images or text are being sent. Similar restrictions exist for devices like email pagers. At first glance, this might seem too restrictive for our purposes. Closer inspection reveals these limitations to be rigid, but less confining than we are initially led to believe.

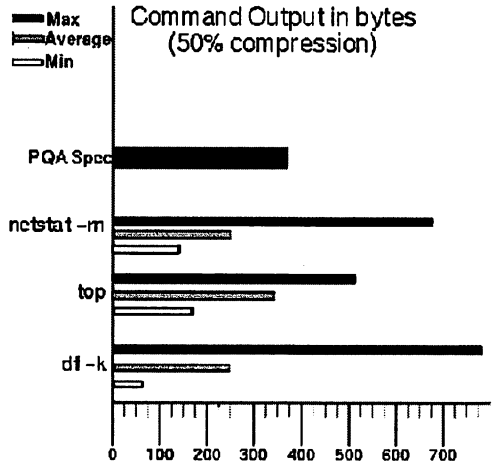


Diagram 1: Transfer sizes.

Here we have a small group of common commands a sysadmin might run to get a sense of the overall health of his or her system and track down potential problems. As can be seen from the graphic, the general trend is that the number of bytes generated by these commands is well within the guidelines recommended by Palm. Even in the maximal cases, none of the data ever reaches even 1KB in size. This gives us a good idea where the basement is in terms of minimum acceptable bandwidth.[4] For faster links, e.g., 9600 bps and above, response time should not be an issue as long as we limit the duration of the connection to what's required to get our work done.

[3]We discount the Metricom Ricochet 128K service as its future is undetermined at this time.

[4]These results were achieved by sending the script

```
df -k|wc -c;top -b -n 1 |
    awk 'NR >6 && NR < 20 {print}' |
    wc -c;netstat -rn | wc -c
```

to several email lists the authors belong to, each with significant subscriber bases of systems professionals, running a variety of Linux,*BSD and Solaris systems and compiling mean, min and max stats on the returned results. No attempt was made to filter out NFS mounts,loopbacks, virtual interfaces, etc. Since this data was purely text-based, we assumed a 50% compression ratio

GEORDI Security

GEORDI Security Basics

With a good idea of the parameters for the raw communications space we were in, we turned our attention to security. We knew that the capability to form an encrypted, preferably authenticated, connection to the remote host would be critical to the success and use of GEORDI. If we could not provide this functionality, the tool would be largely useless save in an academic context of how NOT to implement a GEORDI-like tool. Control of access to privileged commands was equally if not more important, as it is much easier to gain physical control over a handheld system than it is a laptop or desktop. Most privileged remote access is allowed with the assumption that, in addition to whatever electronic security is in effect, the user will still have considerable physical control over the access device. This is not nearly as certain a proposition with handheld systems.

While the PalmOS was a constraint or hindrance in some aspects of this project, there were side benefits to these when it came to security. For a start, the PalmOS is a single-threaded OS which does not provide a shell or user environment one can login to. Limited system memory provides for a very limited number (4 under PalmOS 3.5) of network sockets which must be shared by inbound and outbound connections. The Network Library supports the Berkeley Sockets API, but only through a wrapper layer that is opaque to client and server-side connections. To limit the amount of information available about the systems it contacts, GEORDI does not store usernames, passphrases, keys, etc. in any permanent or temporary databases on the Palm device. These are all held as variables within the program segment itself and are wiped when authentication timeouts occur or when the program exits. This provides protection against them being copied off the PDA through surreptitious beaming [Kingpin] or transferred to another system as a result of a sync operation.

A Daemon-Based Approach

We initially considered a daemon-based approach, where a daemon on the server side would manage client connections and provide both access and privilege management facilities. A GEORDI client would connect to the remote server, present some sort of credentials in the form of a signature or key and request authentication. Upon success, the GEORDI client would be sent a "package" of capabilities outlining what it could do and how long those capabilities lasted.

A Capability would represent a command or series of commands the client was authorized to perform. This would be sent back to the server along with the Key when GEORDI wanted to perform an action there. Upon success, the server would execute the command for the GEORDI client and return the results. The TTL indicated how many times the client could use that capability before it had to request authorization again. Policy indicated whether the key was timestamp or iteration-based. Finally, Server-side Directives gave the server the ability to command and control the client. If a client device was lost or kept trying to perform operations beyond the scope of its privileges, a server directive could be sent to disable the GEORDI Client.

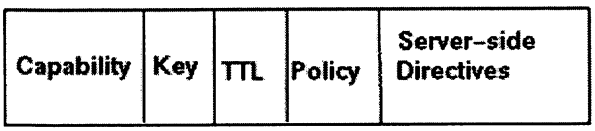


Figure 4: Capability packet diagram.

This was ultimately rejected for traffic as well as security reasons. For one, the worst case of a senior-level person needing access to a wide variety of commands could lead to a situation where a storm of capability data would need to be loaded on a GEORDI client at each startup. This might be acceptable on a high-speed wired network, but over a slow wireless link, it would result in significant startup delays. Additionally, this kind of system could expose detailed information about available system tools and other sensitive server configuration details in the capabilities block if the encrypted session was ever compromised. Finally, there was the ever-present danger of attacks on the server daemon itself and possible exploits therein. A general antagonistic attitude in the security and systems community towards new daemons cemented the cons of this approach in our minds, and we decided to look elsewhere.

The ssh *Approach*

We had avoided an interactive solution using something like ssh because of bandwidth concerns. The size of data that would fly back and forth as keys were exchanged and the CPU time needed to do the necessary cryptographic computations was definitely a concern when aiming for a timely connection. Having seen just how much our other approach could consume, we decided to have another look at it. Ian Goldberg, through the TGssh [GoldbergSSH] application, demonstrated that is was possible to implement a functional SSH client on a handheld system – specifically the Palm Pilot Professional – with a rich variety of encryption algorithms such as 3DES, IDEA, RSA, Blowfish, and others. Our implementation hardware was a more advanced Handspring Visor Prism, so it was encouraging to see that ssh had been implemented on such an early model Palm. Tests of TGssh over a serial link and IRDA at speeds between 19,200 and 38,400 proved that such an application was quite usable from a communications standpoint.

At the same time, we were invited to speak at BayLISA[5] on our work up to that point. We performed an informal survey of the audience and found that the

[5]the local LISA chapter for the San Francisco Bay Area

majority considered password-based ssh to be an acceptable tool for performing remote administration; of that majority, about half admitted to regularly sshing in as root. While this certainly does not reflect the most strict and secure remote access procedures, it does provide insight into the where most place themselves along the security/convenience spectrum.

The choice of ssh as an access method also provided additional benefits with regard to user logging and access to privileged commands. Once logged in, we can make use of any access or privilege control mechanisms already on the system. SSU [Thorpe] held some initial attraction for its use of distinct passwords for privileged operations, per-user command configuration, tight coupling to ssh and account-less login. In the end, we felt that sudo was more suited to our purposes. Since our access of systems is solely from the outside by users of potentially varying authority and responsibility, we had to assume a stance of trusting users as little as possible. Since we were already vulnerable to whatever bugs or holes exist in ssh, the ability in sudo to do more detailed logging and run commands as a user other than root tipped the balance in its favor.

General Security Issues with Handheld Systems

Despite the measures we have employed in the security of GEORDI itself, systems like GEORDI introduce some unique threats to the area of system and network security simply by virtue of the devices they run on. The most prominent of these revolve around the mobility and size of the device that runs GEORDI. The worst case scenario is loss of control or removal of a PDA from the hands of an authorized user. Hardware-based authentication proves only that the device is trusted and known to the system and says nothing about the person using it. Precautions must be taken to guard against Bob grabbing Alice's PDA and running off with it. A simple physical tether like the "The Bond" [Force] would help.

Should this still somehow occur, Alice would quickly call the office to have her server passphrase or keys repudiated. A more forward thinking sysadmin would have also installed a screen-locking program such as "LockMe!" [Witte], "JotLoc" or "GridLock" [GridLock] which locks out the device itself after a certain period of time. Access is regained through the entry of a certain passphrase, as in the case of LockMe! or through the reproduction of a certain pattern as used by GridLock. These are analogous to screen-saver lockouts in the desktop world but are more effective on the Palm. Zero-length boot time and lack of access to the underlying OS contribute to this.

The device can be reset, but the reset process is a physical, multi-step operation; it is accomplished by depressing the pinhole reset switch at the back of the device and subsequently depressing one of the scroll buttons. A simple soft reset does not flush alarms or pending system events, so it should also not disable the lockout program. It is possible for the device to be put into a debug mode, under which access via the serial port is possible, but this has to be explicitly enabled via a Grafitti [Grafitti] stroke during normal operation prior to the reset action. As with any security-related tool or application, it is advisable to test several applications before making a selection for production use. A battery of tests to ensure the utility secures the correct resources and works properly on the model and OS version of the PDA you are using is highly recommended.

Other problems occur as a function of the wireless communications technology itself. Encryption only over the wireless part of the trip from the client to the remote host, proprietary protocols and software, and weak or flawed encryption are all common problems when dealing with wireless networks. The latter two are greater issues which affect both corporate wireless LANs and mobile devices.

We attempt to minimize exposure of the remote system by implementing a timeout of three minutes per connection, after which the user is forced back to the initial login screen to re-authenticate. GEORDI originally operated in a connection-per-command mode to force constant key regeneration, but the overhead of 30 seconds-1 minute per connection over the OmniSky significantly reduced the usability of the tool.

We recognize that there are just some things that can't be done on a system the size of PDA. There will be some cases where GEORDI will not be able to provide the necessary remediation capability needed to fix a particular problem. Some of these may involve network outages, or require the use of a tool or command whose input or output GEORDI is unable to properly represent. Currently, GEORDI does not provide support for access to systems which are inside complex network topologies involving multiple firewalls, DMZs and the like. This could most likely be overcome with some sort of "chat script" system or additional prompting during the connection phase for a user to input a one-time password or additional key. There may also be policy-based reasons within an organization which would preclude the use of something like GEORDI.

Implementation Details

Client Implementation

The handheld device itself is a Handspring Visor Prism, although we use the standard PalmOS 3.5 API, ignoring model-specific features such as color. The wireless device is a Novatel Minstrel S CDPD modem using the Expedite chipset (used for many devices in the "Minstrel" family) with connectivity provided by OmniSky.[6]

Systems constraints within the PalmOS architecture preclude the assignment of a static name to any

[6]See www.omnisky.com for more info

particular device. This holds true for built-in devices such as the onboard serial and IRDA port as well as modules such as the Minstrel. Devices are instead identified at the API level by the service they provide and must be polled each time an application wishes to begin using them.

Functionally, this works out well; An application need only make a request for access to the PPP service to make a connection out to the net. We have successfully used this to connect to target hosts via the OmniSky as well as over the built-in IRDA port. Connections on other PalmOS handhelds using other network devices should therefore work similarly.

Security is handled in two parts. Transaction security is provided by the GEORDI client via the encrypted ssh connection. General device security is provided by the PalmOS security application, although @Stake security services [Kingpin] has warned of a vulnerability in this that allows it to be easily cracked. We have therefore augmented our implementation with a third-party application which restricts the use of the handheld itself as mentioned above. A full survey of available solutions is beyond the scope of this paper, but we found the GridLock and JotLoc [PDABusiness] applications suited to the task. Given the wide range of PalmOS PDAs available for sale, test-driving the available offerings is recommended before settling on a particular application for your own use.

Server Implementation

GEORDI is designed to take advantage of existing user management/access facilities as much as possible, thereby freeing the administrator from having to worry about Yet Another Set Of Configuration Files. If an administrator is already happy with a host's ssh and sudo configuration, he or she has all that is required to allow GEORDI-equipped handheld users to access the system. If not, it will be necessary to set up ssh and sudo first. Below is an extract of the more important aspects of the sshd_config and sudoers files we used in our development and testing efforts.

One thing to note about the sudoers file is that we create a separate User_Alias called GEORDI. It is reasonably easy to embed usernames into the forms that make up the GEORDI UI; therefore a responsible party could make and install GEORDI on a group of corporate handhelds and hand them out to their systems staff. Forcing users to login in under specific accounts lets us exploit the features of sudo more fully than would be possible with ordinary user accounts. Taking this to it's logical conclusion, one could go so far as to set up a chroot jail area and populate it only

```
[...]
User_Alias      PRIMARY=armadilo
User_Alias      LOCALSTAFF=benfell,star,torin,gerbil,rmadillo,kylosohr
User_Alias      GEORDI=geordi,geordi2,eastbldg,westbldg,oncall

Runas_Alias     OP=root,operator

Cmnd_Alias      DUMPS=/usr/etc/dump,/usr/etc/rdump,/usr/etc/restore,\
                      /usr/etc/rrestore,/usr/bin/mt
Cmnd_Alias      KILL=/bin/kill
Cmnd_Alias      PROC=/usr/bin/nice,/usr/bin/renice
Cmnd_Alias      PRINTING=/usr/etc/lpc,/usr/ucb/lprm
Cmnd_Alias      SHUTDOWN=/usr/etc/shutdown
Cmnd_Alias      HALT=/usr/etc/halt,/usr/etc/fasthalt
Cmnd_Alias      REBOOT=/sbin/reboot,/usr/etc/fastboot
Cmnd_Alias      SHELLS=/usr/bin/sh,/usr/bin/csh,/usr/bin/ksh,\
                       /usr/local/bin/tcsh,/usr/ucb/rsh
Cmnd_Alias      SU=/usr/bin/su
Cmnd_Alias      VIPW=/usr/etc/vipw,/etc/vipw,/bin/passwd,/usr/sbin/visudo
Cmnd_Alias      INSTALL=/usr/bin/dpkg
[...]
Cmnd_Alias      MODS=/sbin/lsmod,/sbin/insmod,/sbin/rmmod
Cmnd_Alias      VOL=/bin/mount,/bin/umount,/usr/bin/eject

Host_Alias      RKLABS=miyazaki,otomo,shirow,gainax,ghibli
Host_Alias      AREA66=gecko,iguana,chameleon,komodo,basilisk,tokay,gila
Host_Alias      GEORDI=AREA66

# root and users in group wheel can run anything on any machine as any user
root            ALL=(ALL) ALL
%wheel          ALL=(ALL) ALL

PRIMARY         ALL=ALL

LOCALSTAFF      ALL=PASSWD:DUMPS,KILL,PROC,MODS,VOL
GEORDI          AREA66=NOPASSWD:KILL,PROC,VOL,REBOOT
```

Listing 1: sudoers file.

with those commands and devices that those in the GEORDI group needed to use to do their jobs.

```
[...]
ServerKeyBits 1024
LoginGraceTime 180
KeyRegenerationInterval 600
PermitRootLogin no
IgnoreRhosts yes
StrictModes yes
X11Forwarding no
KeepAlive yes
SyslogFacility AUTH
LogLevel INFO
RhostsAuthentication no
RhostsRSAAuthentication no
RSAAuthentication yes
PasswordAuthentication no
PermitEmptyPasswords no
SkeyAuthentication no
[...]
```

Listing 2: sshd_config file.

GEORDI in action

At base, GEORDI is a forms-based UI wrapper for an RSA/DSA-authenticated ssh connection to a remote host running sudo. The actual over-the-wire commands sent by GEORDI to the remote host are the equivalent of "ssh user@host.subdomain.domain sudo $SOME_COMMAND_STRING." There is a built-in three minute usage window for each GEORDI session, during which the ssh connection remains up. This corresponds to the maximum inactivity time allowed by a PalmOS handheld before it is automatically powered off. Even if this timeout is disabled, GEORDI will return to the main authentication screen after the three minute period and drop its current connection (if any) to the remote host.

On startup, GEORDI reads in a database of approximately 60 UNIX commands. These are a mixture of navigational, network, device and user access commands which exist under most UNIX variants and which are not specific to a particular device or OS distribution. At the same time, GEORDI also loads scripts or commands created in previous sessions by the user with the Command Builder. GEORDI itself can be run without these databases, but the Command Builder feature is disabled if they are missing. It is assumed that the user will have their PATH and other environment variables configured properly in order to use these commands. Not every flavor of UNIX stores its commands in the same directories and providing any sort of pathing information violates the GEORDI security model.

Currently, GEORDI requires that the user type in the FQDN or IP address of the remote host at the start of each session. A username and certificate passphrase is also required to attempt connection. This passphrase is intended for an ssh daemon using RSA/DSA certificates on the remote host. We have tried to make the code as modular as possible, with the expectation that users may need to accommodate something like a SecurID challenge/response system or legacy SSH servers.

If the user authenticates successfully, he or she is presented with a screen similar to that shown in Figure 1. From there the listed processes can be manipulated as necessary by tapping on the appropriate control. In addition to the controls and scripts provided on this screen, a user can construct his or her own custom commands through the Command Builder by tapping the "Build" button.

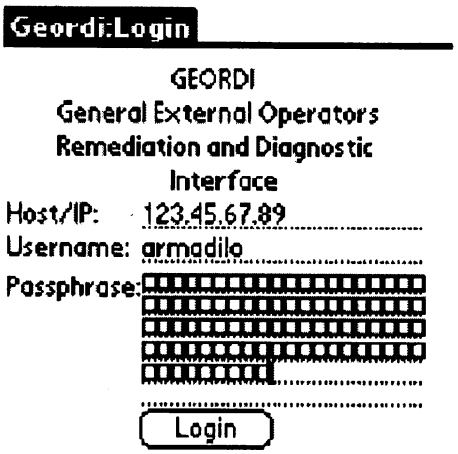


Figure 5: Login page.

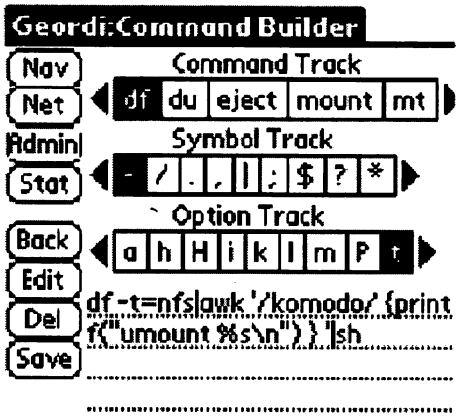


Figure 6: Command builder.

It is here that the aforementioned command bestiary is put to use. New commands or scripts are constructed by tapping on the button containing the desired keyword followed by any additional symbol characters or options in the button rows below the command row. Different sets of keywords are accessed by tapping on the "Nav," "Net," "Admin," or "Stat" commands. Keywords like adduser, mount, df, and the like are organized under "Admin," netstat, ifconfig, route, etc. under "Net" and so on. The option list changes to display only those options for the highlighted command. Saved scripts appear on the drop-down list shown on the main screen.

Configuring the Command Builder

In the version presented here, the Command Builder acquires its data from a companion loader program called GDBI, for GEORDI DataBase Installer. This should be run once for each new GEORDI installation on a particular handheld and then deleted. This approach was taken due to resource constraints in the PalmOS. The program itself is written in C and is easily modified.

The GDBI tool creates a PalmOS record database called "TrkDB" which contains the actual commands shown in the Command Builder as well as metadata on that command's relationship to others in the database. This was done to allow for an eventual feature where commands could be repositioned or reordered according to use frequency or user preference. The basic layout of the Builder is a series of "Tracks" with commands at the top, followed by regex and punctuation characters in the center with command-relevant option characters along the bottom track. Command sections can be changed along the Command track by tapping on one of the "Nav," "Admin," or "Net" buttons off to the left.

Individual records in the TrkDB database have the format in Listing 3.

We expect to have a number of tools available in the near future that will simplify this process down to a CSV-delimited text file suitable for conversion into a PalmOS database, easily installed on the Palm.

What's to stop someone from creating a "script kiddie" command database and using GEORDI to wreak havoc? Indeed, not much. However, the conditions under which this kind of attack would succeed are the same as those involved in most other access exploits. Adherence to best practices security measures such as keeping OS and WKS patches current, disabling deprecated or insecure services, and exercising good user account hygiene are the best defenses against this sort of attack. Additionally, while the GEORDI UI makes a good interface for basic triage and remediation of system problems, it's a lousy interface to crack with and is one use case where the attacker would certainly be better off with something like a laptop.

Tools Used

The GEORDI client was built using the PalmOS 3.5 API, prc-tools 2.0 GCC cross-compiler, pilrc 2.7 form description language and Guikachu form designer. These all freely available for downloading off the net at the following respective locations. You will need the first three to build or work on GEORDI. Guikachu is nice to have for forms design.

- PalmOS API and related documentation: http://www.palmos.com/dev
- prc-tools: http://sourceforge.net/projects/prc-tools/
- pilrc: http://www.pilrc.com or http://www.ardiri.com/palm/pilrc
- Guikachu: http://cactus.rulez.org/projects/guikachu/

Outstanding Issues and Future Work

GEORDI in its current form is not perfect, but we feel it represents a reasonable attempt at a remote system administration tool for handhelds. One immediate missing feature is the ability to access systems which require staging across firewalls or DMZs. Additionally, GEORDI is not yet capable of authentication via hardware related tokens or access paradigms which require some sort of human-entered one-time-password to gain access.

The GUI is a good effort for a device like the Palm with a limited forms-based API. Other systems and devices offer a range of features and functionality and we will probably pursue some of our original UI goals on platforms such as the Compaq iPAQ or one of the embedded Linux platforms. In the other direction, we have received interest from colleagues in porting GEORDI to even smaller platforms such as the RIM Blackberry.

There is work to do in the realm of security and authentication, since widespread use of handhelds, wireless networking, wearables and the like challenge the basic notions of what a host is and force us to reevaluate our threat models. The advent of removable media on handheld devices like the Handera 330, Palm m50x and Sony Clie offers some interesting possibilities in regards to the use of hardware tokens or keys to control operation. A number of commercial security software vendors, such as RSA, have either

```
typedef struct {
    int BuildID;       /* Command ID                                        */
    int LeftID;        /* ID of the Build command to the left of BuildID    */
    int RightID;       /* "    "    "      "       "     right of BuildID   */
    int trackpos;      /* position w/in the Command Track                   */
    int track;         /* Track this command is on                          */
    int section;       /* Section(Nav,Net,Admin) that this command is in    */
    UInt16 CmdTxtLen;  /* Length of the command text                        */
    UInt16 FlagTxtLen;/* Length of the option text                          */
    Char* CmdText;     /* pointer to the command text itself                */
    Char* FlagText;    /* pointer to the flag/option text                   */
} BuildCmdObject;
```

Listing 3: TrkDB database record format.

fielded or announced a port of their technology to the Palm and other handheld platforms.

We welcome any and all contributions and comments on this work. We are particularly interested in hearing from those who might see an application for GEORDI on a organizational or enterprise-level scale.

Conclusions

The current array of handheld devices used to alert sysadmins to problems are capable of assisting with solutions as well. We have stated that failure to exploit these resources will leave us at a disadvantage as staff and machines become more mobile and global in scope and location. We have introduced GEORDI, a PDA-based system using existing best-practice remote access and privilege management software, as a first-order example of using handheld systems to aid in the diagnosis, triage and remediation of system problems. We have looked at other remote access applications and administration tools, noting that tools which conform to look and feel specifications for the target platform are generally more efficient than those that impose an external one. The implementation of GEORDI as presented meets many of the needs of today's mobile sysadmin, and will be developed further as interest increases.

Availability

Further information on GEORDI as well as the software can be found at http://www.geordi.org. GEORDI is made available under the GNU General Public License.

Acknowledgements

This work would have been a much paler effort if it had not been for the assistance of a small mob of individuals. Many dedicated colleagues offered criticism and review of GEORDI at all stages, from initial conception through implementation and this paper.

We are particularly grateful to:

- Jim Dennis, Heather Stern, Dave Benfell, Ken Parker, and David M. Zendzian for testing, evaluation and inspiration with the GEORDI prototype(s).
- Craig Latta for his encouragement and insight throughout the life of the GEORDI project, particularly with regards to the Squeak effort.
- Ian Goldberg for graciously answering all of our(sometimes silly) questions about TGssh and pilotSSLeay.
- Strata Rose Chalup, Cindy Fry, Darren Stalder for helping whip the initial extended abstract into shape.
- William Annis and Ozan Yigit, our shepherds, for readings, edits, commentary, and sticking with us right up to the very end of the process.

Author Information

Stephen Okay is a UNIX geek-of-all trades, first exposed to UNIX in 1986 when working as a student consultant at George Mason University, where he obtained his B.A. 1989. Since then he has performed systems support and programming at MITRE's Center for Advanced Aviation System Development, worked on the SRCSS anti-terrorism mission planning prototype for the Sydney Olympics and did a brief stint at Pacific Data Images as a technical director on the animated feature *Shrek*. He currently consults on mobile computing projects as RoadKnight Mobility Labs. He live in San Francisco, California and can be reached at armadilo@daft.com.

Gale Pedowitz is a developer and UNIX system manager, having worked in such halcyon environs as Sony US Research Labs, TenSquare, and eBreviate. Her research interests include dynamic object systems, human-computer interaction, and network security in the context of system administration. A native of New York City, she currently resides in Portola Valley, California and can be reached at gep@ungeek.com.

References

[Ylonen] Ylonen, T., T. Kivinen, M. Saarinen, T. Rinne, S. Lehtinen. "ssh Protocol Architecture," *IETF Internet-Draft draft-ietf-secsh-architecture-07.txt*, http://www1.ietf.org/ids.by.wg/secsh.html.

[Courtesan] Courtesan Consulting, "sudo," http://www.courtesan.com/sudo/history.html.

[Hall] Hall, B., MarkSpace Softworks, "Online 1.6," http://www.markspace.com

[Ptelnet] de Andrade, M. M., "ptelnet," http://netpage.em.com.br/mmand/ptelnet.htm.

[GoldbergSSH] Goldberg, I., "TopGun ssh (TGssh)," http://www.isaac.cs.berkeley.edu/pilot/.

[PalmOS] Bey, C., E. Freeman, D. Mulder, J. Ostrem, "The PalmOS SDK Reference, Document Number 3003-002," p. 33,769, Palm Computing, Inc., 5400 Bayfront Plaza, Santa Clara, CA. 95052.

[Minenko] Minenko, V., Harakan Software, "PalmVNC," http://www.harakan.btinternet.co.uk/PalmVNC/.

[Richardson, et al.] Richardson, T., Q. Stafford-Fraser, K. R. Wood, and A. Hopper, "Virtual Network Computing," *IEEE Internet Computing*, Vol. 2, No. 1, pp. 33-38, Jan/Feb, 1998.

[Squeak] Ingalls, D., T. Kaehler, J. Maloney, S. Wallace, A. Kay, "Back to the future: The Story of Squeak, a Practical Smalltalk Written in Itself," *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, Vol. 32, Num. 10, October, pp. 318-326, 1997.

[PalmPQA] Brook, James, Scot Stennis, "Web Clipping Developer's Guide, Document Number 3009-002," p. 28, Palm Computing, Inc. 5400 Bayfront Plaza, Santa Clara, CA, 95052.

[Kingpin] kingpin@atstake.com, "PalmOS Password Retrieval and Decoding," @stake Security Advisory (A092600-1).

[Force] Force Technology, "The Bond: Latch for Palm Connected Organizers," http://www.force.com.

[Witte] Witte, R., "LockMe!" http://wwwipd.ira.uka.de/witte/pilot/lockme/.

[Thorpe] Thorpe, C., "SSU:Extending ssh for Secure Root Administration," *Proceedings of the Twelfth Systems Administration Conference (LISA 98)*, pp 27-33, Boston, MA, December 6-11, 1998.

[GridLock] PDABusiness Inc., "JotLoc," "GridLock,"http:// www.pdabusiness.com.

[PilotSSLeay] Goldberg, I., "PilotSSLeay 2.01:SSL for Palm Pilots," http://www.isaac.cs.berkeley.edu/pilot/.

[Stylus] Perlin, K., "Quikwriting: Continuous Stylus-based Text Entry," *Technical Note to the Fourteenth Annual ACM Symposium on User Interface Software and Technology*, November 1-4, 1998 (UIST 98), San Francisco, CA.

[Grafitti] Blinkenstorfer, C. H., "Grafitti," *Pen Computing*, pp. 30-31, January, 1995.

[Costar] Bezemer, J. A., "VT52bis," http://panic.et.tudelft.nl/~costar/hp48.

Macroscopic Internet Topology and Performance Measurements From the DNS Root Name Servers

Marina Fomenkov, kc claffy, Bradley Huffaker, and David Moore – CAIDA/SDSC/UCSD

ABSTRACT

We describe active measurements of topology and end-to-end latency characteristics between several of the DNS root servers and a subset of their clients using the skitter tool developed by CAIDA. We gather a sample of clients for each monitored DNS root server, combine these samples into a common target list and then actively probe these targets and analyze their connectivity. We identify the subsets of destinations that have large latency connections to all instrumented root name servers and discuss their geographical make-up. Our goal is to build an analytical framework for evaluating the optimality of root server placement and its impact on the efficiency of the DNS service. The skitter tool and the methodology we propose can also be used for monitoring the end-to-end performance in large networks and for assessing the optimality of web servers placement in general.

Introduction

The Domain Name System (DNS)

The Domain Name System (DNS) provides name resolution, that is mapping between host names and IP addresses [8]. The DNS is an enormously important service used by virtually all internetworking software, including e-mail and web browsers. In essence, DNS is a distributed database that a) allows local control of its segments; b) makes data in each segment available across the entire network using a client-server scheme [9]. Robustness and adequate performance are achieved by replication and caching.

Programs called name servers constitute the server half of the DNS client-server mechanism; each name server is responsible (authoritative) for its own piece of the database. Clients (resolvers) create queries and send them across the network to name servers. A query process starts when an end user application program contacts a local name server to resolve a host name. If the local name server does not have this name cached, it queries a root server and gets a referral to a name server who should know the answer. The local name server recursively follows referrals until it gets an answer. The most popular implementation of the DNS specifications is the Berkeley Internet Name Domain (BIND) software [10]. The DNS protocol handling requests to name servers and their responses is described in RFC-1034 [8].

The process of name resolution is transparent to an end user, but may contribute significantly to an

Host Name	IP address	Controlling Organization	Location
A	198.41.0.4	VeriSign	Herndon, VA, USA
B	128.9.0.107	ISI	Marina del Rey, CA, USA
C	192.33.4.12	PSInet	Herndon, VA, USA
D	128.8.10.90	University of Maryland	College Park, MD, USA
E	192.203.230.10	NASA	Moffett Field, CA, USA
F	192.5.5.241	ISC	Palo Alto, CA, USA
G	192.112.36.4	DISA	Vienna, VA, USA
H	128.63.2.53	ARL	Aberdeen, MD, USA
I	192.36.148.17	NORDUnet	Stockholm, Sweden
J	198.41.0.10	IANA	Herndon, VA, USA
K	193.0.14.129	RIPE	London, United Kingdom
L	198.32.64.12	IANA	Marina del Ray, CA, USA
M	202.12.27.33	WIDE	Tokyo, Japan

Figure 1: Existing root name servers.

overall delay of establishing connection. Huitema and Weerahandi [11] found that name resolution delays exceeded two seconds in nearly a third of their trial cases. They attributed this poor performance to the flat structure of the domain name space and to name servers overloading. They also found that some of the root servers exhibited unacceptably high loss rates.

In this paper we study the relationship between the geographical distribution of DNS clients and latencies of their connections to the root servers. Our goal is to understand whether the overall performance of the DNS can be improved if existing root servers are re-arranged to bring them topologically closer to certain groups of clients. This problem is also closely related to the question of where additional root servers should be deployed in order to provide maximum service improvement to worldwide Internet users.

The Root Server System

Root name servers (currently, 13 total), are an essential part of the Internet infrastructure. Each name server is responsible for a portion of the naming hierarchy tree that is used to translate host names into IP addresses. The root servers are the first to be queried when a client's name server does not have a requested host name in its cache. A typical load is 5000-8000 queries per second and it appears to scale linearly with traffic [12].

Table 1 below shows the current locations and controlling organizations of the existing root name servers.

The Root Server Selection Advisory Committee (RSSAC) is the DNS root server technical advisory committee for the Internet Corporation for Assigned Names and Numbers (ICANN). One of RSSAC's responsibilities is to provide ICANN with recommendations regarding optimal locations for root name servers (both existing and future ones). RSSAC has asked CAIDA for assistance gathering measurement data to help determine such architecturally strategic locations. The problem is two-fold:

- Are the current locations optimal or is there unnecessary redundancy that can be eliminated?
- Where should ICANN place additional (or relocate existing) root name servers?

We have developed a methodology for identifying and depicting sets of destinations that appear to have consistently large latency connections to all instrumented name root servers. This methodology, if applied at all current and potential future root server locations, can be useful for answering RSSAC's needs.

CAIDA's skitter Measurements

CAIDA uses the skitter tool [1] to actively measure connectivity and performance of the network between root servers and a subset of their clients. skitter sends a small packet to a target host and records the forward IP path traveled and the round trip time (RTT) required to reach that host. CAIDA skitter monitors probe many thousands of destinations several times per day, thus providing data on topology and end-to-end latency characteristics between the skitter host and its target destinations.

We deployed the first root server skitter monitor co-located with the F root name server in August 1999. This monitor probed a target list of F's clients gleaned from a tcpdump on the F's network. In June 2000, we began monitoring the E and L root name servers using corresponding lists of their clients. By October 2000 we had deployed three more skitter monitors: one co-located with the A and J roots, one at the K root, and one at the K peer (RIPE, Amsterdam) servers. Monitoring of the M root in Tokyo began in January 2001. We hope to place monitors at roots B, D, G, H and I in the near future. J is currently co-located with A and so does not require its own monitor. As of August 2001, the C root server had not yet responded to RSSAC's request to host a skitter monitor at their site.

CAIDA also has carried out passive measurements of the root and top-level-domain (gTLD) servers [13]. Brownlee, et al., used two traffic meters located at UC San Diego and captured the number of requests sent to each of the 13 root name servers and 11 gTLD servers, their response time, and the loss rate. They examined the long-term behavior of the name server system and proposed a model of Internet congestion based on these experimental data.

Target Lists

The *DNS Clients* List

Initially, each skitter monitor used its own probe list made up of its own clients. However, if each skitter monitor uses a different probe list, it is difficult to either compare results or to draw global conclusions. We needed to create a global target list that would in some sense stratify the routable Internet (IPv4) address prefix space. We proposed to achieve such representative coverage by including a destination from each routable IP prefix in a new target list. In September 2000, when we built our current ***DNS Clients*** , there were nearly 90,000 globally routable prefixes [2].

We created the September 2000 DNS Client destination list from two sources:

- combined client lists from each root server (49,374 addresses)
- addresses from CAIDA target lists for other projects (8,944 addresses)

From the data gathered statically at each root server location by tools such as tcpdump or cflowd, we culled 49,374 IP addresses that belonged to different routable prefixes. If many client IP addresses were from the same routable prefix, we chose the host that we observed querying the root server most often.

Though an imperfect weighting of the client population, this method allows us to focus on a single client list that reasonably reflects global Internet connectivity to the instrumented root servers.

To increase prefix coverage, we added addresses from our other skitter destination lists [3]. These addresses, which now comprise about 15% of the DNS Clients list, are probably not local name servers but rather hosts on a network that either have or may in the future have a name server. Therefore, data quantifying performance of root servers for these future potential DNS-root client networks are also relevant for our project.

As we deploy additional skitter boxes at other root servers and obtain their local client destination lists, we will merge these addresses into the global list. In particular, if a particular prefix is represented by an IP address from our other non-DNS-client lists and we find another host in that prefix in a local DNS client list, we will replace the initial placeholder host with this latter IP address.

Characteristics of the *DNS Clients* List

The current DNS Clients list contains more than 58,000 IP destinations, covering 8406 origin Autonomous Systems (ASes) and 184 countries. Table 2 shows the top ten top level domains, origin ASes, and countries represented in this target list. Note that some of our lists have deliberately tried to include hosts from as many different countries as possible at the expense of proportional representativeness of per-country connectivity. This bias may linger in the DNS Client list to the extent that it draws on those other lists.

We determine the top level domain with a reverse DNS lookup to find the host name associated with the IP address. Note that of the 58,000 IP addresses on the current list, more than 21,000 do not have PTR records and therefore their host names are unresolvable. We determined the origin AS for a given IP address via the BGP routing tables from Route-Views [2] snapshot from 4 April 2001, by finding the longest matching routing prefix and then noting which AS advertised that route. CAIDA's NetGeo tool [4], [14] estimates the geographical location of the IP addresses. This procedure may be imprecise, particularly for hosts at an ISP's site. If a site does not have a DNS LOC record and does not include geographical hints in a host naming scheme, then NetGeo relies upon whois entry which usually refers to company headquarters. This, in turn, is not necessarily reflective of the actual geographical location of the host. Figure 3 shows the distribution of targets by continent.

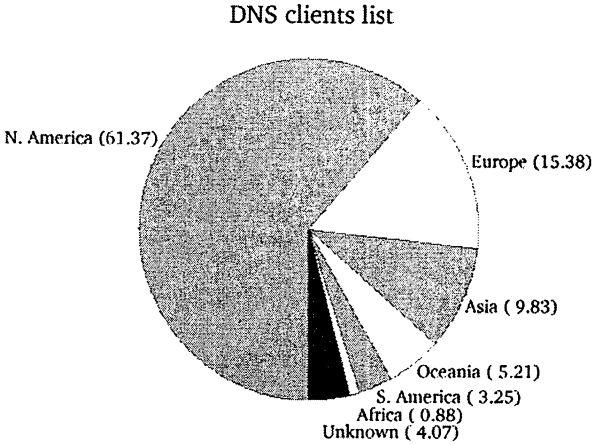


Figure 3: Geographic distribution of destinations in the DNS Clients list by continent. The numbers in parentheses are percentages.

skitter Host	Start Date
A root	October 5, 2000
E root	September 27, 2000
F root	November 2, 2000
K peer	October 5, 2000
K root	September 19, 2000
L root	September 19, 2000
M root	January 9, 2001

Figure 4: Dates we started monitoring the DNS Clients list.

Top level domains		Origin ASes		Countries	
com	11345	AS 701, ALTERNET	1660	USA	31172
net	8697	AS 1, BBN Planet	577	Canada	3276
au	1929	AS 7018, AT&T	546	Australia	2645
edu	1763	AS 3561, Cable & Wireless	538	unknown	2373
jp	1376	AS 2914, Verio	472	Germany	1681
ca	1212	AS 1785, Applied Theory Corp.	472	Japan	1285
org	969	AS 1239, Sprint	467	U. K.	1061
de	891	AS 1221, AARNET	428	France	981
us	854	AS 2200, INRIA-Rocquencourt	358	Mexico	803
mil	673	AS 2907, SINET	335	South Korea	794

Figure 2: "Top Tens" of the DNS Clients list.

We have used the combined DNS Clients list since the fall of 2000 on the skitter monitors at the root server locations. Figure 4 shows the exact dates when we began probing this list from each root server skitter host.

Issues with the *DNS Clients* List

There are important advantages to using the same list on each root server skitter monitor. A common list serves as a yardstick against which we can compare characteristics of the different root server networks. Stratifying the IPv4 space by probing as many routable prefixes as possible yields a representative macroscopic view of Internet topology from root server locations. Note that the list is also geographically diverse and thus allows us to explore the dependence of RTT on the geographical location of a destination. We recognize that the geography is among the primary factors determining the latency of Internet connections and have explored geographic and topological correlations with performance in more detail [5, 6].

By measuring the same destination list simultaneously from each root server, we can identify a group of destinations that show high latency from all monitored root servers. High latency could be due to the bottleneck bandwidth along the path to the target host often at the last hop or due to an unfavorable topological location of the target host relative to the root servers. If a set of such high latency destinations clusters either geographically or topologically and does not have systematic regional bandwidth problems or other political constraints, it might be a candidate region that merits a new root server.

An apparent disadvantage of monitoring the merged DNS Clients list is that we cannot use these data to decide how well a particular root server responds to its own specific clients. This problem arises due to an internal BIND load balancing feature [10]. There is a code in all recent versions of BIND that causes name servers to intelligently select among alternative queryable root servers. BIND measures the round trip time for each of multiple answers from candidate servers and sorts these values into groups based on the observed values of RTT. It directs subsequent queries to servers in the closest group, in a round robin fashion. As a result, a name server close in performance terms to a particular root server will query that server most often, only occasionally querying other servers that are further away. However, we know which destinations in our list were frequent clients of which particular root server, and can use local subsets of the DNS client lists to study individual server-specific issues.

Results and Analysis

In this section we analyze two sets of data, 30 days long each. We collected the first set of traces between December 1, 2000 and December 30, 2000, and the second one between March 6 and April 4, 2001. In both sets, we used the same DNS Clients list on all DNS root server skitter monitors. However, there are two important differences between the sets.

- Monitoring of the M-root server had not begun as of December 2000. We obtained traces on the monitors co-located with the A, E, F, K, K-peer, and L root DNS name servers. In March 2001, the M-root monitor was operating, but the L-root monitor experienced some local connectivity problems and was temporarily disconnected. Therefore, the second set consists of traces obtained at the A, E, F, K, K-peer, and M root DNS name servers.
- itskitter software, which considerably increased probing efficiency. The daily number of probes sent by each monitor in March data is 15-60% higher than in December data.

Measurements

Each monitor probes destinations in the DNS Clients list between seven and 13 times per day. The frequency of sampling depends on processing capabilities of skitter hosts and also decreases somewhat when the network is congested. In each cycle through the list, skitter probes usually reach between 31,000 and 33,000 destinations. The number of unique destinations reached during weekdays in our March, 2001 measurements ranges from 36,000 to 33,000 and dips during the weekends (see Figures 5 and 6).

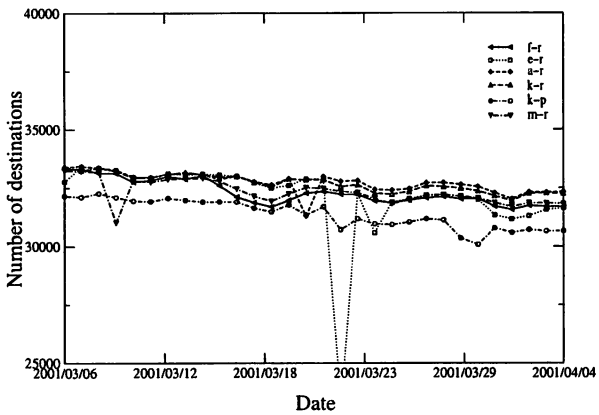


Figure 5: Average in each probe cycle (March, 2001).

Comparison of the daily number of replying destinations in the December 2000 and March 2001 data confirms an overall declining trend apparent in Figure 6. We found that the loss rate of target hosts from the DNS Clients list is (1.8±0.2)% per month. Destinations may stop replying to skitter ICMP probes for a variety of reasons (firewalls, internal changes of IP addresses in businesses, etc.). We plan to expurgate non-replying destinations from our target list and to replenish it with new destinations again gathered statically from the root servers. Our goal remains to represent each globally routable IP prefix in the updated DNS Clients list. Note that the number of such

prefixes continues to grow and is now greater than 100,000.

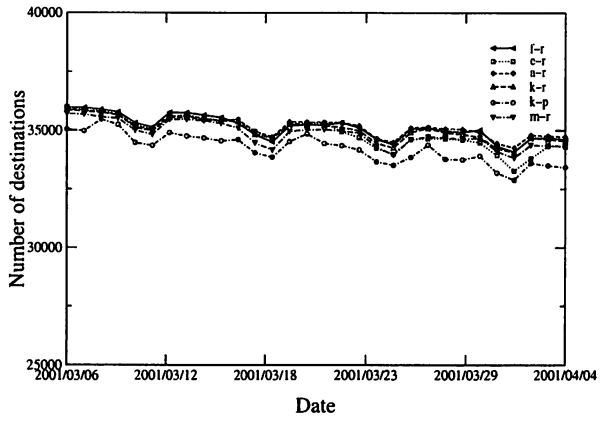


Figure 6: The number of unique destinations replying per day (March, 2001).

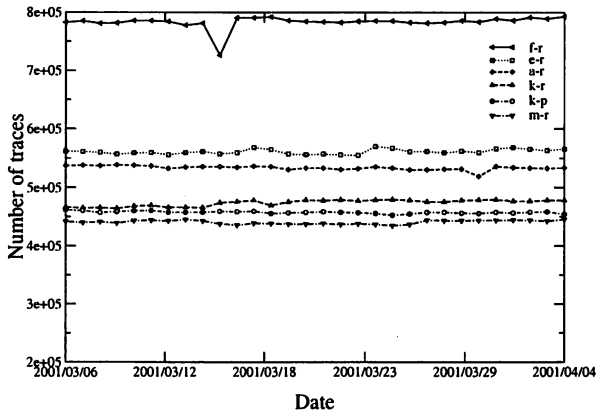


Figure 7: Number of probes sent by each skitter host (March, 2001).

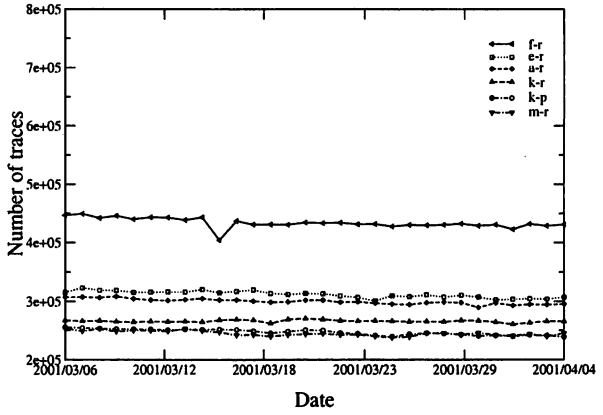


Figure 8: Number of replies collected (March, 2001).

skitter monitor records RTTs to replying destination hosts. If intermediate hops along the path failed to answer skitter probes, but the final destination still responded, we included such a ***responding but incomplete*** path in our analysis. Each CAIDA skitter host sends between 450,000 and 800,000 probe packets per day and collected between 250,000 and 450,000 replies during the course of our measurements (see Figures 7 and 8).

We analyze two metrics of connectivity: hop count and round trip time from the root name server to the hosts in the target set. IP (layer 3) hop count is a natural connectivity metric that characterizes topological proximity of a skitter source to its target set of destinations.

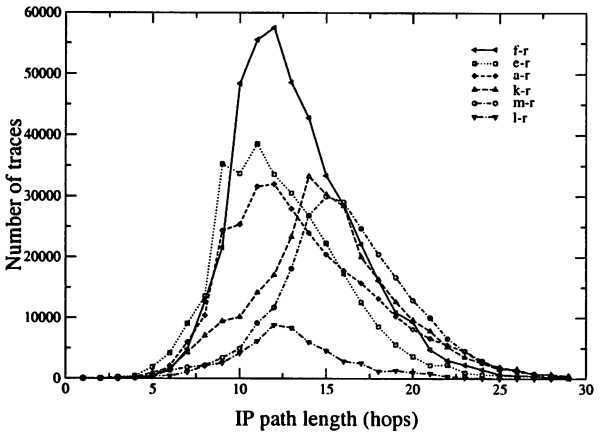


Figure 9: The distribution of IP path length, April 4, 2001.

Figure 9 shows IP hop count distributions for the six root server skitter monitors that ran the DNS Clients list on April 4, 2000. E, F and L are in California; A is in the Washington DC area, K is in the United Kingdom, M is in Japan. The curves are not normalized; the y-axis shows the actual number of probes (highest for the F monitor, lowest for the L monitor). The L skitter monitor had consistently fewer complete probes per cycle than others, suggesting problems with its local connectivity. Since February 2001 its connectivity has been intermittent.

The x-axis value that corresponds to the peak of the distributions (the mode) depends primarily on two parameters: the geographical distribution of the targets in the list and the connectivity of the skitter source. The peak positions for A, E, F, and L root server monitors (all in the US) indicate that they are near the edge of their local networks and/or near a major exchange point. The IP hop count distributions for the K-root monitor (in the United Kingdom) and for the M-root monitor (in Japan) are shifted to the right, implying that these monitors are further away from most of the DNS Clients list destinations (unsurprising, since the list is heavily dominated by North American destinations). The IP hop count distribution for K-peer is rather similar to the one for K-root and is not shown in the figure.

Values of RTT (i.e., latency), observed in skitter data depend on the geographic and topological position of the skitter monitor with respect to the destinations it probes. Latency also depends on the conditions of the Internet along paths to those destinations

(congestion, routing instabilities, etc.). We found that RTTs exhibit significant diurnal and weekly variations. Internet paths tend to be less congested on weekends, when RTTs to all destinations drop considerably from their weekday values.

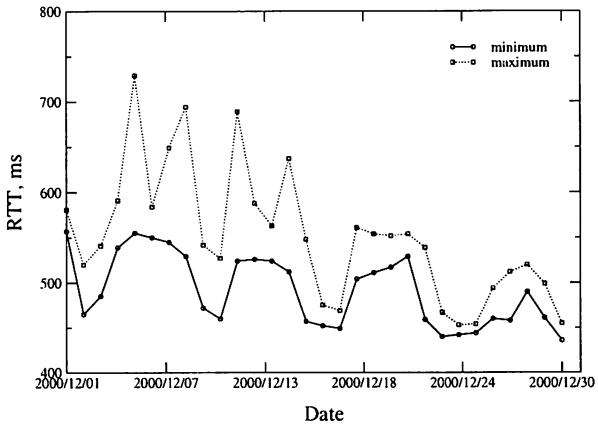


Figure 10: The 90th percentile of RTTs for the F-root skitter monitor in December, 2000.

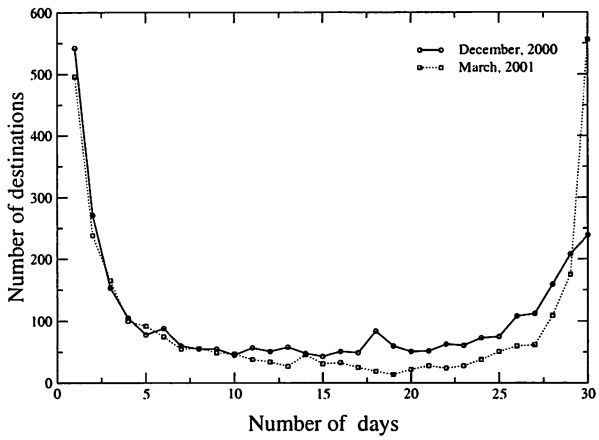


Figure 11: The persistence of large latency destinations.

Large Latencies

Clusters of hosts that have particularly large latencies as measured from all root name server skitter monitors suggest a potential deficiency in the current Internet infrastructure. A large latency could be due to the location of the roots relative to the client or due to the local connectivity of the client. In order to identify target hosts that have high latency from the set of currently monitored root servers, we analyzed the daily distributions of RTTs seen by root server skitter monitors.

In each probe cycle we consider the value of RTT to a destination as large if it is above the 90th percentile of the overall RTT distribution for this cycle. Typically, large RTTs are longer than 500 ms, sometimes as high as 1000 ms. It is necessary to deal with the RTT distributions in each cycle separately because of large diurnal variations in the state of networks (more congested during the business hours, less congested at night). Figure 10 illustrates the daily and monthly variability of the 90th percentile values observed by the skitter monitor co-located with the F-root server. The graph shows that the maximum value of the 90th percentile of RTT distributions observed in a day of the data can be as much as 1.5 times higher then the minimum value. A significant decrease in overall network latency during the weekends and/or holidays is also clearly seen.

We then define a destination as having ***large latency*** during a given day if on that day it had large RTTs in at least half the cycles on all root server monitors. We increase the statistical robustness of our results by aggregating them on a monthly basis to ignore transient problems that are repaired relatively quickly. Figure 11 shows how many destinations had large latency and for how many days during the thirty day periods starting on December 1, 2000 and on March 6, 2001. The first (left-side) maximum in both curves is due to the random variations in connectivity that caused a number of destinations to have large latencies for a day or two. The second (right-side) maximum reflects destinations that consistently have large latency on every (or almost every) day during the corresponding period.

We then selected the destinations that had large latency for at least 24 days during the period of measurements (974 of them in December, 1051 of them in March) and mapped them to their origin ASes and to their countries/continents. Figure 12 shows these data organized by origin AS, that is, the AS advertising routing information to the Internet. Each AS listed represents more than 1% of the large latency subset. Of the 7882 origin ASes represented in the DNS Clients list, 282 ASes were associated with the large latency destinations we found in December 2000. In March 2001, the corresponding numbers were 8406 and 316.

Figure 13 displays the same data sorted by country. Of the 184 countries represented in the DNS Clients list, 105 and 114 contained high latency destinations in December 2000 and in March 2001, correspondingly. Each country listed in Figure 13, contributes more than 1% of this large latency subset.

We see the following changes between the December and March analysis:

- Thailand, Jordan, Georgia, Costa Rica, Brazil, and Fiji contributed less than 1% to the large-latency subset of March 2001, while Bangladesh, Turkey, Bulgaria and Nigeria contributed less than 1% to the December subset.
- The number of large latency destinations in India, Romania and South Africa has decreased between December and March by 20%, 36% and 36%, correspondingly.
- The number of large latency destinations in Ukraine more than doubled, and in Chile it increased almost five-fold.

Figure 14 compares the make-up by continent of the DNS clients list and of the two large-latency subsets. Percents shown include all the data, not just the values listed in Table 15. It is clear that the general geographical pattern of large latency destinations remains nearly the same in both samples. As expected, monitoring our target list from the M root location caused the number of large latency destinations in Asia to decrease: from 274 (28.1% of the subset in December) to 247 (23.5% of the subset in March). At the same time, the number of large latency destinations in South America increased: from 166 (17.0%) to 260 (24.7%). There are two possible explanations for this. The connectivity to South America may have deteriorated since the first sample. Alternatively, the apparent degradation we see may result from not having the L root server monitored in the second sample.

To differentiate between these two possibilities, we analyzed the December 2000 data set with paths from the L skitter monitor excluded. The total number of large latency destinations increased from 974 to 1050, with the number for each continent increasing proportionally. If it were the L-root name server site that primarily provided lower RTTs to South American destinations, then exclusion of these data would cause an unproportional increase of this continent share in the large latency subset. We do not observe this. We thus hypothesize that the increase observed in the March 2001 data is not caused by the lack of the L-root data, but rather reflects an actual (although, possibly temporary) change in connectivity.

Figure 14 shows that Africa, Asia, and South America IP addresses account for over 60% of the observed large latency destinations, but less than 14% of the total client list. The exact numbers are: 15% versus 0.9% for Africa, 26% versus 9.8% for Asia, and 21% versus 3.3% for South America (averages of the both data sets). The African destinations have the highest relative increase across the two data sets. Does this mean that a new root server should be placed there?

Before we can draw any conclusions about the cause of the large latency responses, we must measure the bottleneck bandwidth to these large latency destinations to ensure that the last hop across a slow modem link is not the primary cause of the delay. It is

	December 2000			March 2001		
	# of targets	# high latency	% high latency	# of targets	# high latency	% high latency
AS 3741, Internet Solution	92	55	60	102	49	48
AS 4755, APNIC	204	49	24	174	22	13
AS 7545, APNIC	128	30	23	138	30	22
AS 2905, TICSA-ASN	38	24	63	38	21	55
AS 2277, ECUANET	35	19	54	32	21	66
AS 7633, APNIC	28	19	68	29	19	66
AS 10530, Interpacket Group	101	18	18	72	25	35
AS 11127, NetSat Express	39	18	46	28	15	54
AS 6140, IMPSAT ARGENTINA	59	16	27	55	14	25
AS 6471, ENTEL CHILE	55	15	27	55	16	29
AS 6453, Teleglobe	54	14	26	68	18	26
AS 3132, Red Cientifica Peruana	23	14	61	24	10	42
AS 8143, Publicom	29	13	45	26	15	58
AS 7087, COLOMSAT	25	13	52	24	10	42
AS 9241, APNIC	15	12	80	15	11	73
AS 2018, UNINET-ZA	44	30	68			
AS 4621, APNIC	21	17	81			
AS 2614, RIPE	18	14	78			
AS 1239, SprintLink	398	10	3			
AS 6429, AT&T Chile Internet				137	73	53
AS 3255, RIPE				34	24	71
AS 7418, PROVDESERV				26	14	54
AS 5511, RIPE				33	14	42
AS 9471, APNIC				25	12	48
AS 7473, APNIC				41	10	24

Figure 12: Origin ASes of large latency destinations.

also possible that incompleteness of our monitoring infrastructure somewhat skews the results. For example, we see that the proportion of large latency destinations in Asia decreased when we included the data obtained with a skitter box co-located with the M root server in Tokyo. The deployment of skitter monitors near other root servers and augmenting our target list with their clients is likely to change at least some of the results reported here.

Conclusions and Future Work

CAIDA's skitter measurements can be used with local client lists to analyze topology and performance characteristics of the network between a root name server and its typical clients. Other placement issues, such as distance to the edge of the local network, peering relationships, choice of upstream transit providers, are visible from the graphs provided by the daily summaries generated automatically from each skitter monitor's data [7].

skitter measurements with the combined DNS clients list can identify clients that have large latency destinations, measurements need to include monitoring from all 13 root servers. We hope that the remaining root operators will soon be hosting skitter monitors, to each of the current root server locations being monitored. To minimize bias with respect to large latency

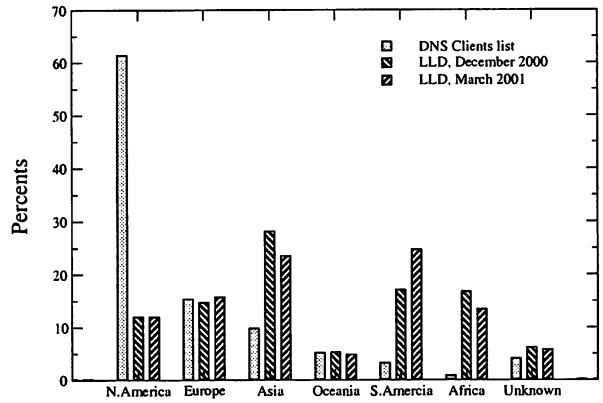


Figure 14: Percentages of the entire target list and of the two large latency subsets by continent.

Continent	Country	# targets	December 2000		March 2001	
			# high latency	% high latency	# high latency	% high latency
Asia	India	382	94	25	75	20
	Indonesia	165	35	21	36	22
	Pakistan	88	18	20	21	24
	Russia	437	14	3	15	3
	Thailand	315	25	8		
	Jordan	31	11	35		
	Georgia	15	10	67		
	Turkey	175			15	9
	Bangladesh	13			11	85
Europe	Romania	377	86	23	55	15
	Ukraine	185	30	16	71	38
	Bulgaria	203			14	7
North America	USA	31172	71	0	74	0
	Costa Rica	35	12	34		
South America	Ecuador	90	34	38	40	44
	Chile	375	30	8	142	38
	Argentina	592	27	5	19	3
	Colombia	213	25	12	21	11
	Peru	88	19	22	17	19
	Brazil	411	14	3		
Oceania	Australia	2645	29	1	23	1
	Fiji	13	10	77		
Africa	South Africa	268	124	46	79	29
	Nigeria	12			10	83
unknown	—	2373	60	3	60	3

Figure 13: Countries of large latency destinations.

which do not interfere with name server operation at all. We would like to thank those root operators who have hosted our monitors.

Further examination with other tools are needed to determine the primary cause of the large latency. If we can eliminate sites that have low client bandwidth at the end of the path, we will have a subset of destinations that can guide the selection of new sites for root servers.

We suggest that any site under consideration for a root server could run a skitter monitor for at least six months using the augmented DNS clients list as well as the large latency clients list to determine performance characteristics of the network between the proposed name server site and its potential clients. Issues such as distance to the edge of the local network, rich peering relationships, and adequate upstream transit from multiple providers are good prerequisites to choosing potential sites.

Note that placement of new root servers should also take into consideration prospects of advancing Internet growth. Therefore more than just current large latency destinations should be considered in selecting potential new root server sites. Empirical data gathered from macroscopic performance measurements across large segments of the IPv4 topology provide valuable input into the decision process. In addition, a model that simulates geographic patterns of the Internet use should be developed, tested and applied to predict future DNS needs and trends in root server usage.

We believe that our methodology and results are also applicable to many common problems such as optimizing web server placement or monitoring the performance of a particular network. In the latter case, if the bandwidths are known, then monitoring the RTTs to hosts of the network with the extit{skitter} tool (or similar) will immediately identify a poorly connected subset of the network.

Acknowledgment

We are grateful to the root server operators who have cooperated in installation, configuration, and data collection at their sites. The authors thank Evi Nemeth and Daniel Plummer for their help with finishing this manuscript. We highly appreciate the insightful comments and constructive suggestions made by anonymous reviewers and by our shepherds Mark Burgess and Frode Sandness. This work was supported by DARPA NGI Cooperative Agreement N66001-98-2-8922 and by the NSF ANIR grant NCR-9711092.

Author Information

Marina Fomenkov is an Internet traffic researcher for the distributed Cooperative Association for Internet Data Analysis (CAIDA), and a research scientist at the University of California's San Diego Center for Astrophysics and Space Sciences. She is developing algorithms for analyzing skitter active analysis traffic data. Marina received her M.S. in Experimental Nuclear Physics from Moscow Institute of Physics and Technology, and her Ph.D. in Engineering and Data Processing Systems from Moscow Space Research Institute. Reach her electronically at marina@ipn.caida.org.

kc claffy is a principal investigator for CAIDA, and a resident research scientist based at the University of California's San Diego Supercomputer Center. kc's research interests include Internet workload/performance data collection, analysis and visualization, particularly with respect to commercial ISP collaboration/cooperation and sharing of analysis resources. kc received the Ph. D. in Computer Science from UCSD in 1995.

Brad Huffaker serves as a technical manager of several tool development and traffic analysis efforts at CAIDA. He has developed several Java based tools for the visualization of Internet topologies. Brad's current focus is as technical lead on skitter project. Brad graduated from UCSD with the M.S in Computer Science.

David Moore is the Co-Director and a PI of CAIDA. His research interests are high speed network monitoring, denial-of-service attacks and infrastructure security, and Internet traffic characterization. He also led the development of NetGeo, an automated tool that maps IP addresses, domain names, and Autonomous Systems numbers to geographic locations.

References

[1] "Caida Network Measurement Tool skitter," http://www.caida.org/tools/measurement/skitter/.

[2] D. Meyer, "University of Oregon Route Views Project," http://www.antc.uoregon.edu/route-views/.

[3] "Caida skitter Destination Lists," http://www.caida.org/tools/measurement/skitter/lists/.

[4] "Caida Network Measurement Tool netgeo," http://www.caida.org/tools/utilities/netgeo/.

[5] Huffaker, B., M. Fomenkov, D. Moore, E. Nemeth, and k. claffy, "Measurements of the Internet Topology in the Asia-Pacific Region," *Proceedings of INET00*, Yokohama, Japan, The Internet Society, 2000.

[6] Huffaker, B., M. Fomenkov, k claffy, and D. Moore, "Macroscopic Analyses of the Infrastructure: Measurement and Visualization of Internet Connectivity and Performance," *Proceedings of PAM2001, A workshop on Passive and Active Measurements*, Amsterdam, Netherlands, RIPE NCC, April, 2001.

[7] "Caida skitter Daily Summary," http://www.caida.org/skitter_summary/main.pl.

[8] Mockapetris, P., "RFC 1034: Domain Names – Concepts and Facilities," Oct. 1987.

[9] Albitz, P., and C. Liu, *DNS and BIND*, O'Reilly and Associates, 1998.

[10] "Bind website." http://www.isc.org/products/BIND/.

[11] Huitema, C., and S. Weerahandi, "Internet Measurements: The Rising Tide and the DNS Snag,"

Proceedings of 13th ITC Specialist Seminar, Monterey, California, 2000.

[12] Rood, H., "What is in a Name, What is in a Number: Some Characteristics of Identifiers on Electronic Networks," *Telecommunications Policy*, Vol. 24, pp. 533-552, 2000.

[13] Brownlee, N., kc Claffy, and E. Nemeth, "DNS root/gtld Performance Measurements," *Usenix LISA XV paper*, December, 2001.

[14] Moore, D., R. Periakaruppan, J. Donohoe, and k. Claffy, "Where in the World is netgeo.caida. org?," *Proceedings of INET00*, Yokohama, Japan, The Internet Society, 2000.

DNS Root/gTLD Performance Measurement

Nevil Brownlee – The University of Auckland, New Zealand and CAIDA, SDSC, UC San Diego
kc claffy – CAIDA, SDSC UC San Diego
Evi Nemeth – University of Colorado and CAIDA, SDSC, UC San Diego

ABSTRACT

The Internet Domain Name System (DNS) is an essential part of the Internet infrastructure. Each web site or email lookup involves traversing a tree-structured distributed database to complete the mapping from a hostname to an IP address. The root and top level domain (TLD) nameservers form the highest level of authority over the Internet naming hierarchy, and are thus potentially involved in reaching any and every URL or email address we seek. We use passive measurements to analyze performance of these critical nameservers from a client network's viewpoint.[1]

We use NeTraMet meters on a university campus to take passive measurements of DNS response time, request loss rate and request load to the root and gTLD (generic top level domain, e.g., .com, .net, .org) servers.

From these measurements we produce strip charts that are useful for day-to-day monitoring of one's Internet connectivity, since they reveal changes in network behavior on paths between one's local network and the global servers without the need to actively inject traffic into the network. We are developing a monitoring tool to produce such plots in near real time.

Introduction

DNS, the Domain Name System, is responsible for translating between hostnames used by people and corresponding IP addresses needed by software. The data for this mapping is stored in a tree-structured distributed database where each nameserver is authoritatively (responsible) for a portion of the naming hierarchy.

The DNS protocol [1] is a request/response protocol based on UDP transport. BIND [2], the Berkeley Internet Name Domain System, is the reference implementation used by most sites. Local nameservers contact root nameservers and then traverse the DNS tree until their query arrives at a server that has the answer. Root servers provide referrals to nameservers for country-code domains (e.g., .au, .uk, .us), and generic top-level domains (gTLDs, e.g., .com, .net, .org). A referral is basically an answer that says "I do not know the address of yahoo.com, but here is the address of the .com servers who *will* have that information."

BIND has a built-in load balancing mechanism when a query results in more than one answer, as happens for a query for the root zone (13 answers) or gTLD zones (11 answers). When BIND gets a response with multiple answers, it keeps track of the Round Trip Time (RTT) to each of the servers in the answer and sorts them into bands. BIND will then round robin among the servers in the band with the lowest RTT, but also periodically reduces the RTT stored for each server in other bands. These far away servers will thus eventually be in the closest band, BIND will then query and resort them based on a more recent RTT. This technique causes BIND to usually choose the servers that are nearest in terms of latency, and spread the load across them, but also to re-calibrate itself in case a server's performance was anomalous rather than representative.

As well as load balancing, BIND caches replies it receives from other nameservers. Each answer carries a Time To Live (TTL) value, telling the local nameserver how long it may locally cache that answer. For frequently used domain names BIND will usually have a cached answer. As well as improving response time to the user, caching dramatically reduces the load on upstream nameservers in the tree. In other words, caching makes the DNS scale.

Figure 1 shows the location of the global root and gTLD nameservers. Each has a one-letter name, i.e., A, B, ..., M, and is identified by city. The servers are not evenly spread throughout the world – they are clustered on the east and west coasts of the US. For example, the A, C, D, G, H and J roots and the A and G gTLDs are all located in the vicinity of Washington, DC. Similarly, there are four roots and four gTLDs on the US West Coast, near San Francisco and Los Angeles. The roots are run by individual organizations, the gTLDs are all administered by Network Solutions.

[1]Support for this work is provided by DARPA NGI Contract N66001-98-2-8922, NSF Award NCR-9711092 'CAIDA: Cooperative Association for Internet Data Analysis,' and The University of Auckland.

The root and gTLD nameservers are crucial to the Internet infrastructure. Until mid-2000 the root servers also served the gTLDs. As the growth of the Internet increased the workload on the roots, the .com, .net, and .org domains were moved off the root system onto a separate layer of 'global top level domain' (gTLD) servers run by Network Solutions. There are 11 of these gTLD servers. While the hardware and operating systems of the root servers is architecturally diverse, with several manufacturers and operating system vendors represented, all 11 gTLD servers are identical IBM AIX machines, distributed around the world.

We use a traffic metering tool called *NeTraMet* [3] to examine behavior of the root and gTLD servers from a client site's perspective. We run *NeTraMet* on top of an OC12 (622 Mb/s) link monitor, CoralReef [4], that sees university traffic via an optical splitter on the wide area circuits connecting the university to the Internet. We measure request rates, response times (RTT) by matching response and request packets, and request loss rates.

A recent paper on Internet performance includes DNS name resolution latency measurements. Huitema & Weerhandi [5] found that the end-to-end latency of name lookup exceeded two seconds in 29% of the cases. Our measurements do not show nearly this amount of delay, however they are not end-to-end measurements, but rather edge of campus to root servers or gTLD servers and back. Perhaps the discrepancy in the observations derives from a slow or congested Internet connection at the point of their measurements or is due to efficient caching on our campus. At the time of Huitema & Weerhandi's measurements, the root servers also served the gTLD domains and thus may have been overloaded.

In the next section, we describe our measurement methodology, and then discuss the client-side measurement results.

Measurement Methodology

Our measurements are passive, which means we observe the traffic flowing by, rather than actively injecting any traffic into the network. As it observes DNS packets, our traffic meter performs data reduction, such as computing response times. The meter writes the reduced data to 'flow data' files for later analysis.

Our campus network is connected to the commodity Internet and to three research/academic networks. In May 2000 we installed one OC3 (155 Mbps) traffic meter, our *commodity* meter, to monitor the commodity Internet link. Our campus network topology is such that there is no single point where a traffic meter can see traffic for the entire campus on all four external links. Instead, in January 2001, we placed a second OC3 meter at a boundary point within the campus network where it could see traffic for the inner part of the network, including a large fraction of the traffic to the commodity link and two of the three research/academic links. We call this our *edu* meter. In July 2001 our network configuration changed. We responded by reconfiguring our *edu* meter to monitor an OC12 link at a point in the new topology where it could see traffic to and from all four external links. Initial measurements in June and November 2000 used our *commodity* meter; the January 2001 measurements use both meters and the July 2001 measurements use only our OC12 *edu* meter.

We also have data measured at a site in New Zealand; unfortunately the request rate there was too low to provide reliable estimates of response times or request loss rates for the global root servers. Even so, our New Zealand data correlates well with that collected at San Diego.

Our traffic meter [3, 6, 7, 8] is an open source implementation of the IETF's Realtime Traffic Flow Measurement (RTFM) architecture for network traffic

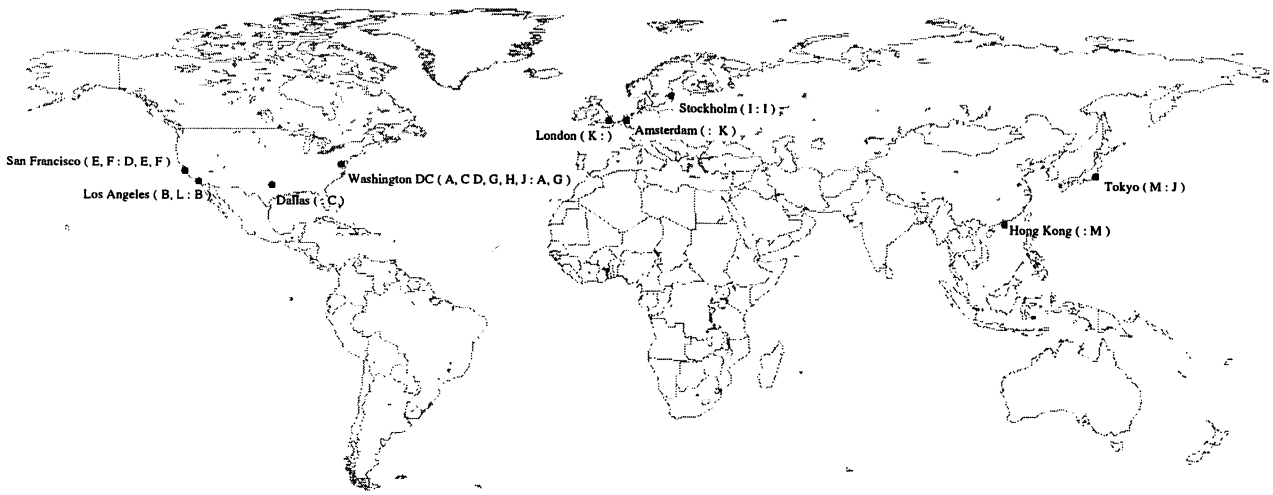


Figure 1: Location of root and gTLD servers. Each city lists servers (root: gTLD) using their one-letter names. Note the uneven geographical server distribution, with high concentrations on the US East and West Coasts.

flow measurement [9, 10, 11]. It is a highly configurable, passive, real-time link measurement tool, and has been extended to use the CoralReef library [4] to read packet headers from either a live network or from a trace file.

We used the traffic meter to measure response times for the global DNS root and gTLD nameservers by configuring it to capture DNS request packets and their corresponding response packets. The meter maintains queues of packets for each source-destination address pair and uses the ID field of the DNS header to match each incoming response with a queued request.

We also configured the meter to measure the number of 'unanswered' DNS requests, i.e., requests for which our meter did not capture a response. We use 50 bins for our distributions, with a logarithmic response-time range of 7 to 700 ms; response times above 700 ms are counted in a single overflow bin. Requests are considered unanswered if they do not receive a response within 10 times the highest bin value, i.e., 7 seconds.

We filter our captured data to include nameserver traffic travelling in both directions. To ascertain the degree of asymmetric routing affecting our global DNS traffic, we configured our meters to count DNS

- requests that receive a matching response. We use these to compute response times, which include the server response time as well as the network travel times.
- 'unanswered' requests. We compare these with the matched requests to produce plots of the request loss rate.
- 'unrequested' replies, i.e., response packets for which our meter did not previously capture a corresponding request.

DNS requests may be unanswered for several reasons:

- A request may come from a host with an invalid IP address, for example one from 'private' address space [12]; a reply cannot be routed back to such a host. Our meter is configured to ignore requests from addresses that do not belong to prefixes announced by our campus.
- A query may be badly formed. For example DNS queries should only contain a single question; queries with multiple questions violate the DNS protocol specification; the root servers will drop such requests. Some root servers see a

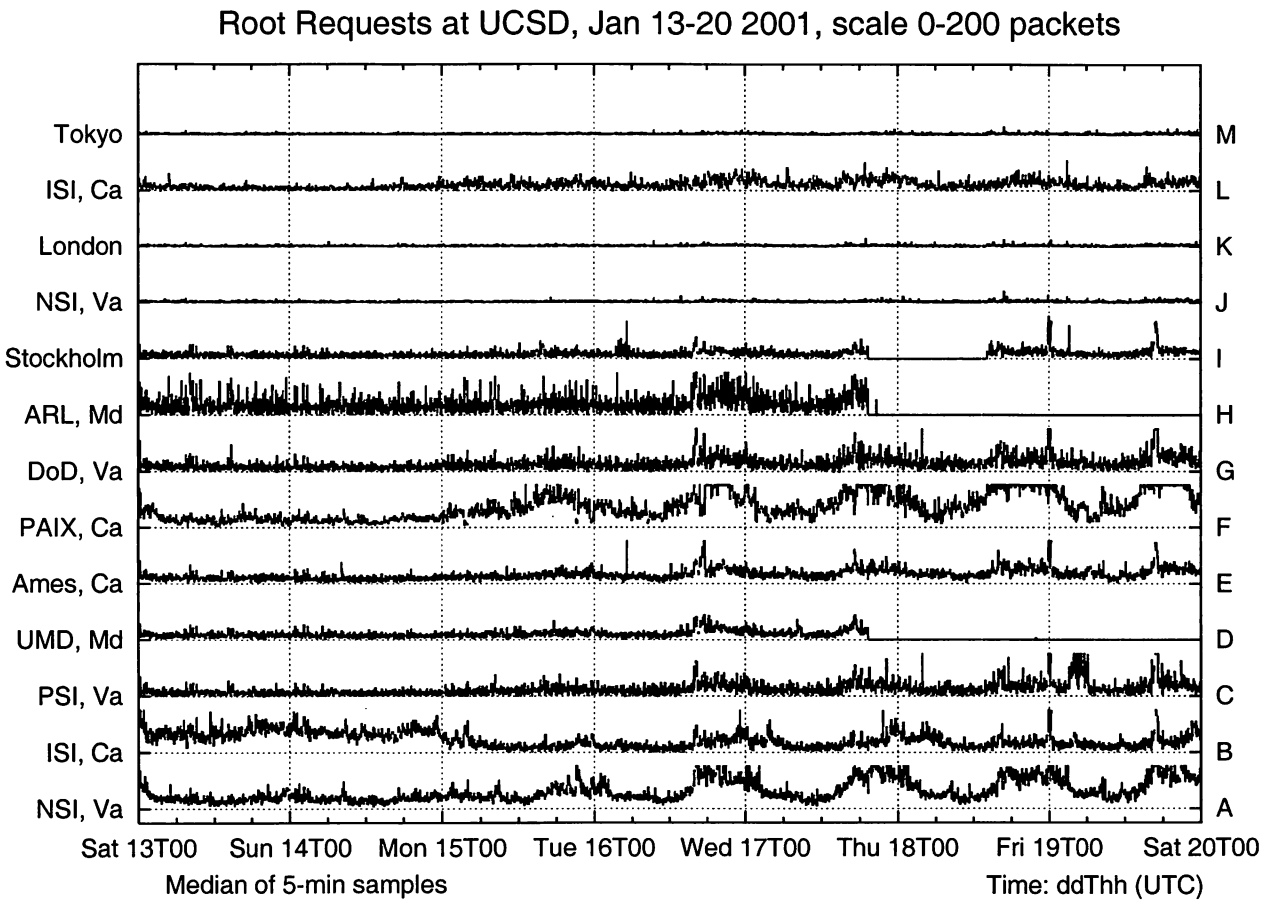


Figure 2: DNS requests to root servers reflect our local BIND's view of the servers. Here BIND is favoring A and F roots (we see clear diurnal variations), and sending few requests to J, K and M. BIND stops sending requests to D, H and I at 2000 on Wed 17 Jan, indicating that connectivity to them has been lost.

significant number of queries whose DNS header indicates 256 queries in the packet, when in fact there is only one – a Microsoft byte-order bug affecting the header field [13].

- Asymmetric routing may cause a request and its response to travel via different networks, so that a meter may see only one of these two packets. As indicated above, our meters are placed and configured so as to minimize this problem.
- A packet may be dropped on the way to the nameserver, at the nameserver, or on the way back.

Overall, our request loss rate plots give a good indication of losses in the network or at the nameservers.

Finally, we measure the total data rate in each direction on our commodity link. We compute rates every ten seconds and use this interval to build distributions for traffic rates in both directions. We were curious about traffic rates since the link in question was a rate limited (20 Mb/s) ATM OC3 circuit and we wanted to verify that the rate-limiting was not affecting our request loss rate measurements.

We collected flow data from the traffic meters at 5-minute intervals, and computed the corresponding median values. For each of the 13 root nameservers and 11 gTLD servers, we collect the number of requests sent, the response time, and the loss rate. Not all nameservers use BIND; older versions of the Microsoft servers ask only the A servers (first in the list of nameservers in a referral), resulting in disproportionately higher query rates to the A server. We discarded data intervals (of five minutes each) if there were less than 10 queries in the interval, i.e., too few queries to produce statistically defensible data for that interval.

We do see data to each of these servers due to internal algorithms in the BIND software that spread the load when there is more than one possible choice of nameserver to contact. Observing the spread of requests across the root or gTLD servers provides us with BIND's view of the state of those servers, and provides a sensitive indicator of changes in a server or its network paths.

Experimental Results

We have data from four time periods: June 2000, November 2000, January 2001 and July 2001. Results from our *commodity* meter in June 2000 showed high

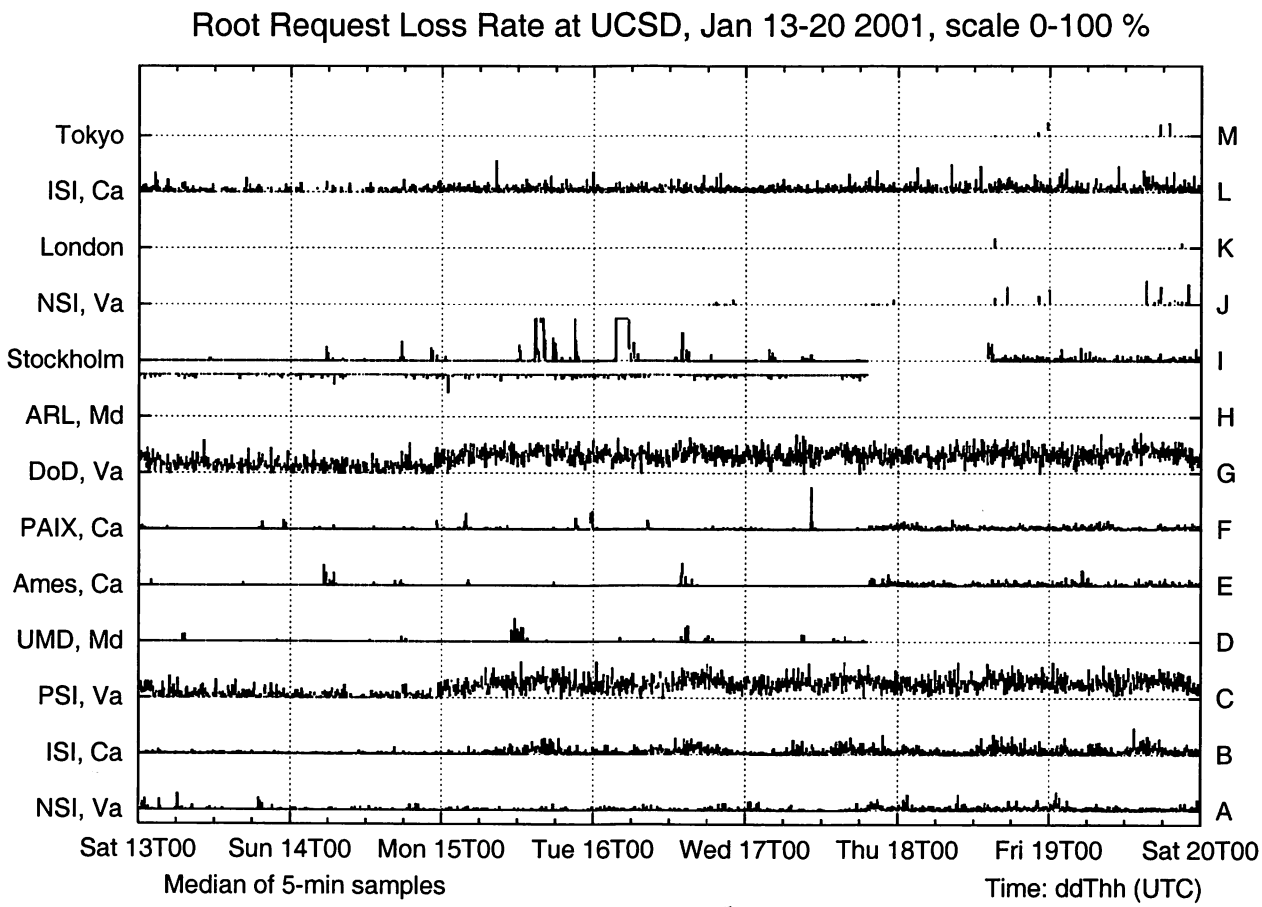


Figure 3: Heavily loaded root servers (C, G and B) show high and variable loss rates, especially during weekdays. H root has almost 100% losses until connectivity fails at 2000 on Wed 17 Jan. Other servers show short loss spikes, indicating network losses.

unanswered-request or unrequested-reply rates, primarily due to asymmetric routing. To overcome this idiosyncrasy in our topology, we chose a set of local nameservers whose requests and responses travel via the commodity link, and configured the *commodity* meter to filter out DNS packets to and from other local nameservers. This filtering reduced the 'unanswered' and 'unrequested' counts to negligible levels.

Due to local routing policy, our *commodity* meter does not see DNS request or response packets for two of the roots. In January 2001 we began using our *edu* meter, which sees traffic for all the root and gTLD servers, and higher DNS traffic rates than for the *commodity* meter. Early experience with the *edu* meter showed a significant reduction in unanswered-request rates, but it also showed 'unrequested-response' rates of the same order. This indicates that our meter was seeing asymmetric routing, possibly via the fourth (unmetered) external link.

We changed to our OC12 *edu* meter in July 2001. Data collected since then shows negligible levels of 'unrequested' responses. We believe that our request loss plots are reliable for November 2000, and from July 2001 onward. Our January 2001 data covers three consecutive weeks, for the other periods our data covers periods of only two to four days. We will concentrate on the January data to describe request rates, request loss rates and response times, and refer to the other data to see trends over time.

We use two types of graphs: strip charts and server performance '4-plots.' In the strip charts we plot data for either the 13 root nameservers or the 11 gTLD servers on a single figure with time on the x-axis and stacked narrow strips on the y-axis, each strip with data for a single server. The time period is a full week (Saturday to Friday) so workday variations, weekends and holidays are clearly visible (for example, the A and F servers in Figure 2). The server performance 4-plots show median response times, request loss rates, total query rates, and median overall load on the rate-limited commodity link.

Times on the plots are indicated in the format ddThh [14], i.e., day, T, hour of day. We use UTC times – they are eight hours ahead of the local time zone, i.e., 8 a.m. San Diego time is 1600 UTC. This, together with the Saturday to Friday format rather than the more typical Sunday to Saturday format causes the weekday peaks to appear shifted to the right.

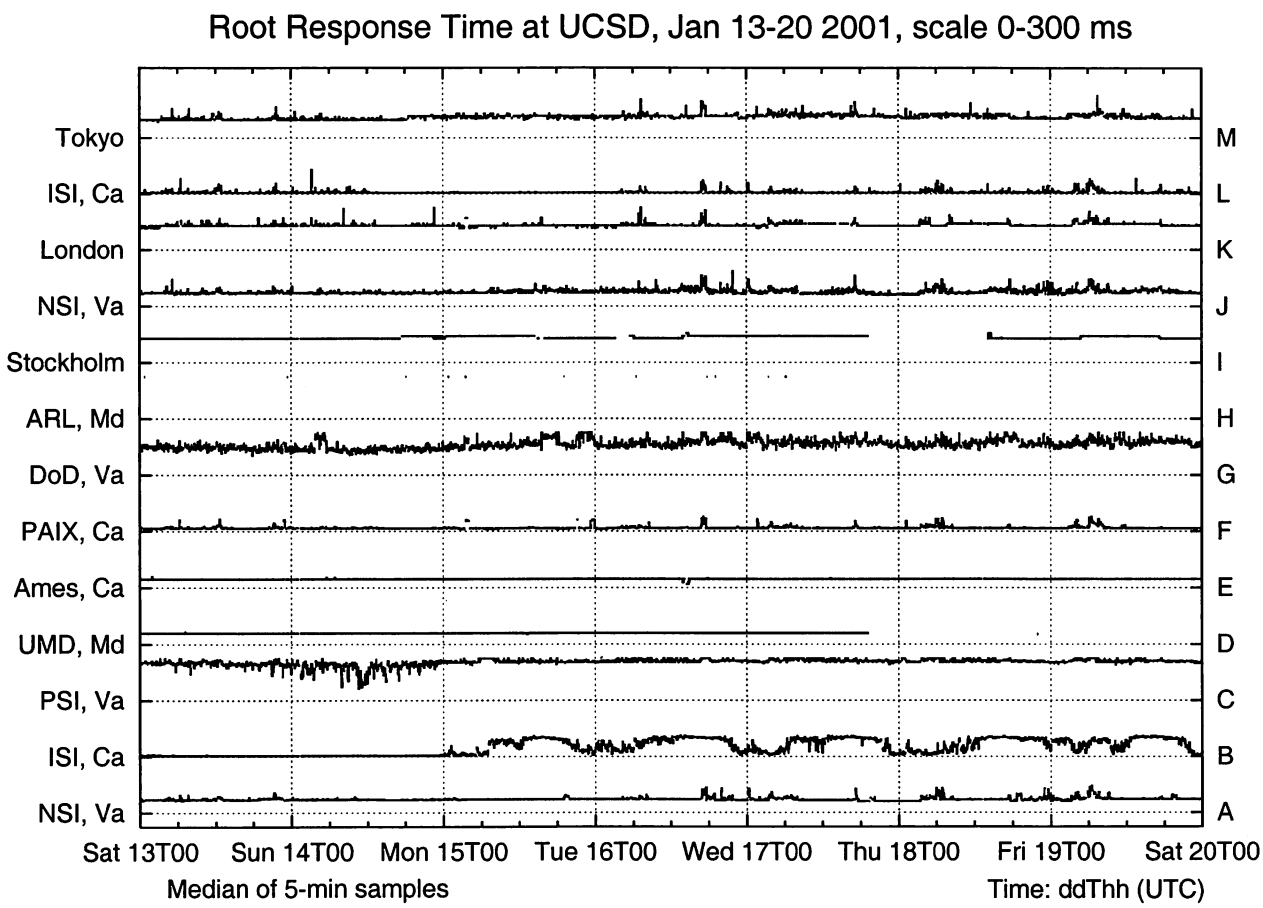


Figure 4: DNS root response time traces are normally flat with occasional spikes (A, E and F). Overloaded servers have persistently high response times, e.g., H (only occasional dots plotted), C and G. B is close to overloading, having good response times during the weekend, but poor response (with diurnal variations) during the week.

Strip Charts: Root Nameserver Performance

Our first goal in this study was to develop an understanding of how the global root and gTLD nameservers behave as viewed from our campus. We produced strip charts for the 13 root servers (A to M) and the 11 gTLD servers, plotting the number of requests, median unanswered request percentage, and median response time. Values are all plotted on the same vertical scale (shown in the title at the top of the graphs), with high values clamped at the maximum.

Figure 2 shows DNS requests to the root servers for the week January 13-20, 2001, as seen by our *edu* meter. Monday, January 15 was Martin Luther King Day, a holiday in the US; request levels are lower on this 3-day weekend than during the week. Flat tops on the curves correspond to times where the query load to that server was more than 200 per five minute interval as seen for F root during weekdays. Requests are spread across all servers, with A and F the most heavily used. BIND uses its own round trip time measurements to determine closest servers and uses those more frequently, as described in the first section.

We lost connectivity to the D, H and I roots at about 2000 on January 17; connectivity to I was restored about 1500 on January 18. Our normal routes to these servers all go via the same research network, so the plots indicate a failure in that network. The flat sections of the request plots for D, H and I (Figure 2) show that BIND reacted by dramatically reducing its request rate to those servers, waiting for them to reappear.

Figure 3 shows the percentage of unanswered requests to the root servers; we omit time intervals with too little data (<10 requests) since they are not statistically valid. Request loss rates are usually a few percent, but these losses are not generally visible to users because BIND (at the local nameserver) masks them by caching earlier responses, by retrying requests using other servers (rather than resending to the same server), and by preferring servers that respond quickly.

The H server, which had loss rates above 90% until we lost connectivity to it late on 17 January, was particularly bad. Response times for the H root are the worst on Figure 4 – they appear on the plot as occasional dots at about 360 ms.

Each server also showed request loss rate spikes in the 10% to 25% range. Four of the root servers are

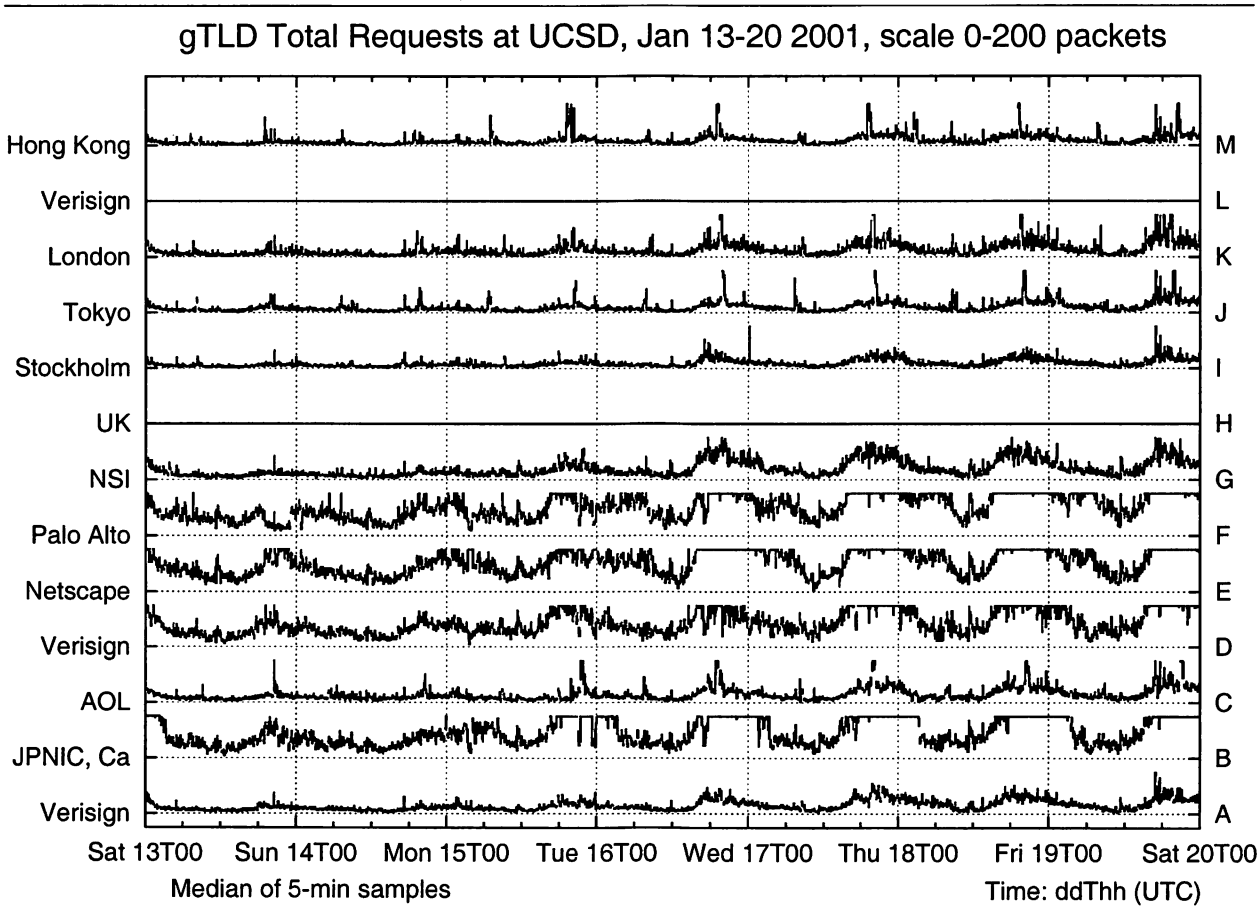


Figure 5: Our local BIND sends many more requests to gTLD servers than to roots. BIND favors those servers located on the US East Coast (B, D, E, F), producing clear diurnal variations during weekends as well as weekdays.

located in California, another six are in the Washington DC area. *traceroute* shows that packets to servers in each of these two groups follow common paths until they are close to the servers. Spikes at different times for different servers within these groups suggest congestion at points beyond the common segments of the paths. The additional delay may derive from the network near the servers, or at the servers themselves. Losses that occur across several servers at the same time indicate congestion near the measurement meter. The C and G servers show higher and quite variable loss rates, typically around 70%. Our routes to C, F and G root go via our commodity Internet link, then through different ISPs to each server. Their loss patterns and response times differ noticeably, suggesting that the observed variability for C and G indicates either congestion at or near those servers or, more likely, overloading of the server machines themselves.

Figure 4 shows observed response times for requests to reach root servers and responses to return. Roots A, F, E, K and L showed fairly flat response times, with short variations suggesting periods of congestion. Roots C and G have high response times and are highly variable, reflecting their loss behavior.

B root shows very good performance on Saturday 13 and Sunday 14 January, with response times varying between 7 and 12 ms. Over that weekend B has the lowest root response time, (the next lowest was F, 18 to 22 ms). As Figure 2 shows, BIND recognized this, and sent most of its DNS requests to B during that time. After the weekend, however, B's performance deteriorated – its unanswered request rate increased and its response time increased dramatically. BIND responded by sending requests to other root servers instead. The data is consistent with a server (B, in this case) being unable to handle its request load.

Figure 4 also shows roots A, I and K with small but marked (10 to 20 ms) extended steps in response time, lasting 12 hours or more. To determine whether these changes were due to route changes, we examined data collected by CAIDA's topology mapping project [15] during the week shown in Figure 4. Forward path topology traces were available for paths between our university and the A and K roots, collected at intervals of about 50 minutes. In that week (13-20 January 2001) we saw few forward path changes for the A root, but many forward path changes for one or two hops within the path from our

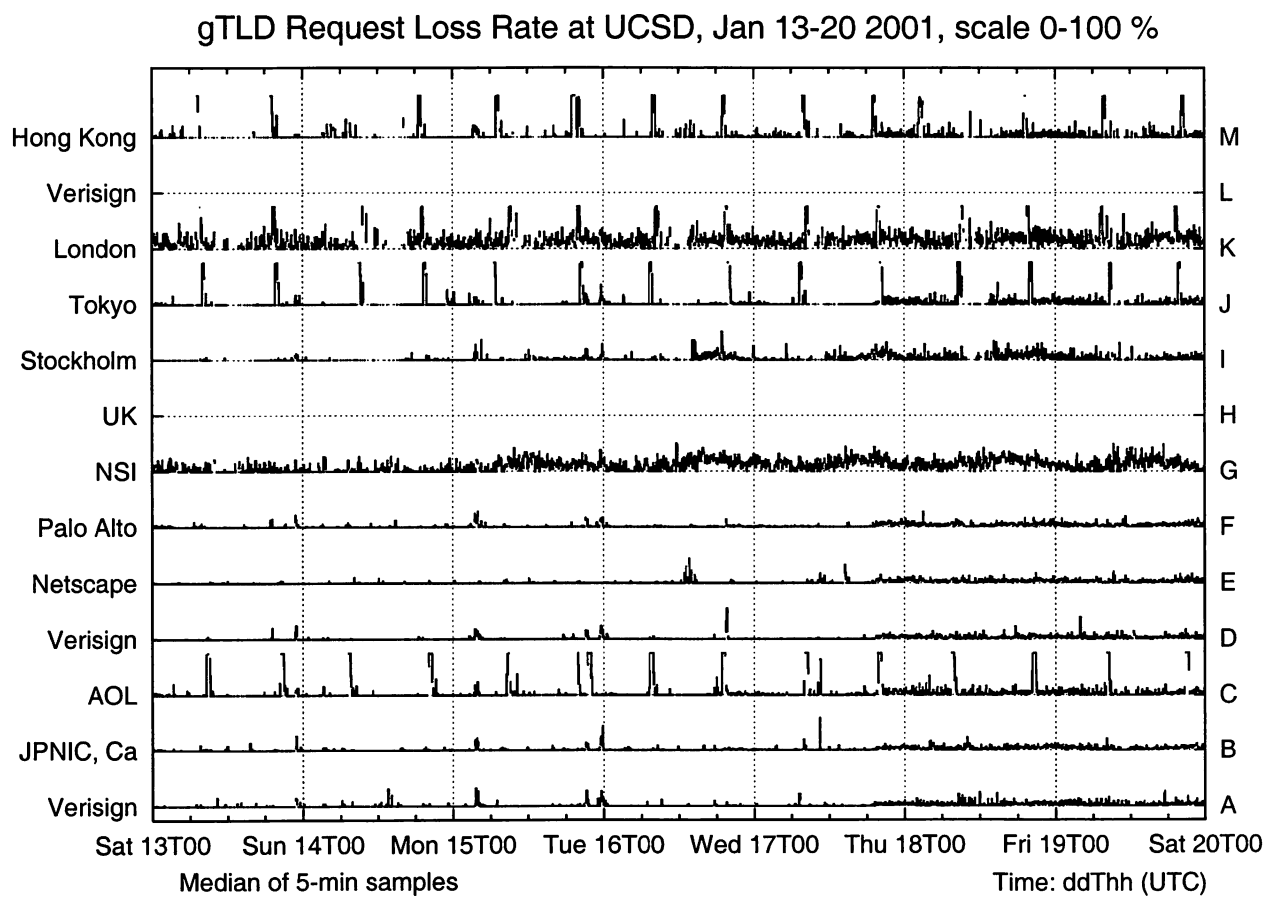


Figure 6: gTLD server loss rates show a regular pattern of spikes about every 12 hours for the C, J, K and M servers. These spikes appear when the servers are reloading their zone tables. The G and K servers show high background loss rates; since their response times remain good (Figure 7), these losses indicate network problems.

university to the K root. These path changes did not appear to be correlated with changes in round-trip time. Distribution bin sizes limit the precision of our 5-minute median response times, nonetheless they lie within one bin of the observed skitter round-trip times. We believe that the steps in our response time traces may simply be artifacts introduced by our choice of bin sizes.

Strip Charts: gTLD Nameserver Performance

Figures 5-7 exhibit generally similar behavior to the root server behavior discussed above. Most requests went to the B, D, E and F gTLDs; these four gTLDs are all located in California, so BIND favors them because they have lower response times from our campus than other gTLD servers. There were considerably more gTLD requests than root requests, reflecting the size of the domains involved.

Figure 6 is similar to Figure 3, except that it shows 5-minute median loss rates for the gTLD servers instead of the roots. We observed a regular daily pattern of 100% loss during a 30 minute period twice a day as certain gTLD servers reload the .com zone (>2 GByte). This pattern was also visible from New Zealand's meter, in spite of the substantially lower DNS request rate there.

Figure 7 shows the observed response time for gTLD servers. Response time spikes at different times for different servers suggest congestion, most likely away from the measurement point. Losses that occur across several servers at the same time usually indicate congestion near the measurement meter.

The very clear response time steps, for F at 2000 on Sat 13 Jan and for C at 1600 on Mon 15 Jan, indicate route changes in paths to or from those gTLD servers. Figure 5 shows that BIND reduces its request rate to F during the route change. We do not see a similar change in request rate for C because it is nearer to the US West Coast, and BIND was sending comparatively few requests to it. In each case normal routing and response time were restored after a few hours.

4-plots: Correlations between DNS metrics

Figure 8 compares behavior of the A and F gTLD servers from Saturday 27 to Wednesday 31 January, 2001. Routes to the A and F gTLD servers normally go via our commodity Internet link; this link's total load (i.e., all packets, not just DNS) is shown in the bottom chart of Figure 8. Both inbound and outbound traffic rates are often close to the commodity link's rate limit of 20 Mbps. The plot shows median

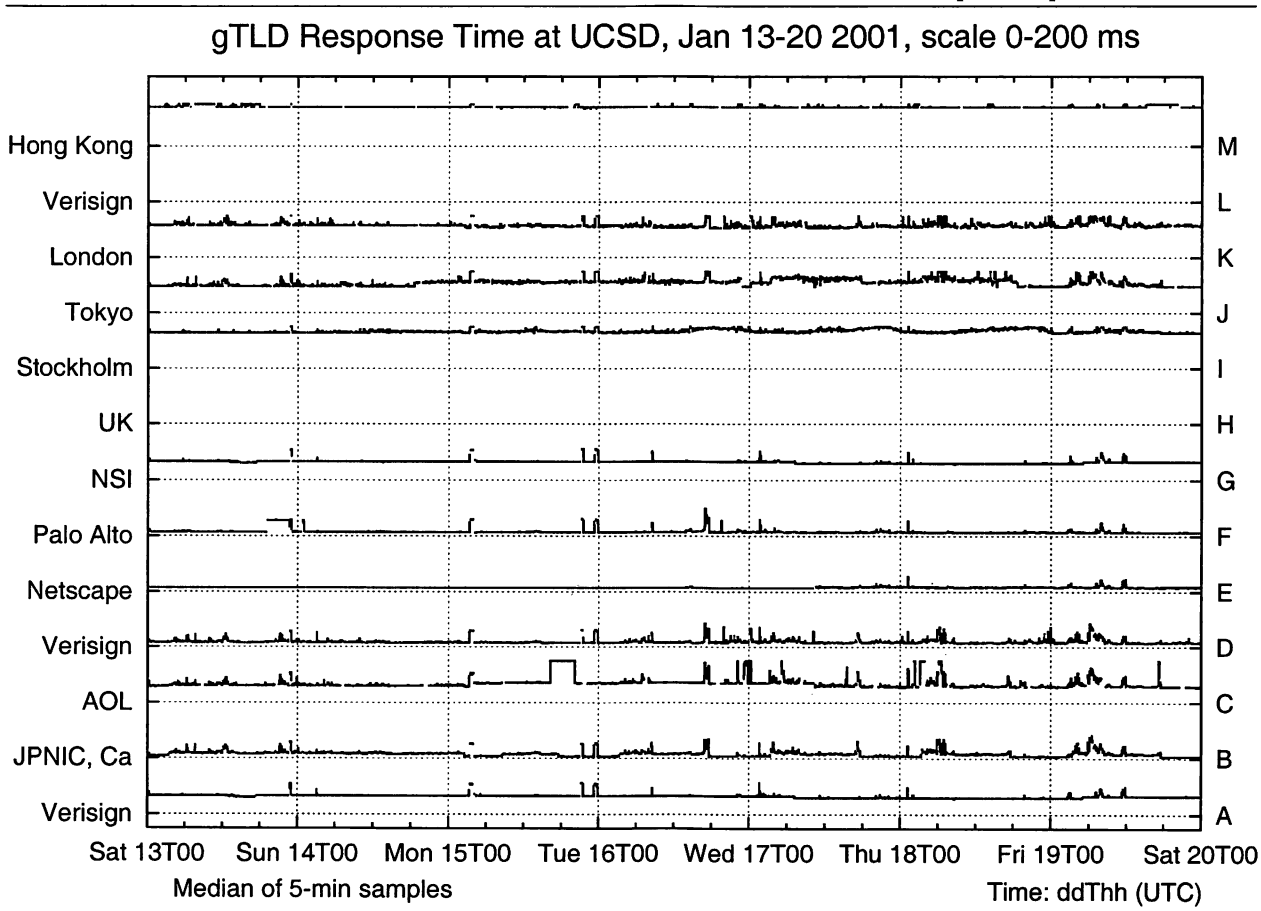


Figure 7: gTLD response times are generally steady, with short spikes common to many servers indicating network congestion. Pronounced steps, e.g., for F at 2000 on Sat 13 Jan, C at 1600 on Mon 15 Jan, indicate route changes.

traffic rate for 10-second intervals, i.e., the link was carrying more than this for half of the 10-second intervals. The link is clearly saturated (and discarding packets) for tens of seconds at a time.

Periods when the link is saturated often produce increases in response time, for example on the 28th from 0130 to 0300, the 29th from 1800 to 1900. For these two periods there is also a marked increase in the unanswered request rate, suggesting increased packet loss. However, we also see periods when the request loss rate increases with no change in the response time (e.g., from 1400 to 1600 on the 30th), or when response time increases with no change in the request loss rate (e.g., from 2300 to 2400 on the 29th). We plan to investigate correlations between these metrics in a future study.

During the four days shown in Figure 8, A gTLD's median response time drifted slowly up from about 90 ms (Saturday) to about 110 ms (Monday) and back to about 100 ms (late Tuesday). The obvious steps in its response time plot (grey line, upper subplot) are caused by our choice of binning intervals.

Overall the two servers behave similarly, showing spikes in response time at identical times. Since they are located on opposite coasts of the US, the fact that paths to both of them have problems suggests congestion close to our meter. However, they do behave differently in some ways, for example the spikes around 0300 on Sunday 28 January – which affect A but not F – and the slow drifts in A's response time. Our 4-plots provide an effective way to compare server behaviors.

Observed Behavior for Some Global Nameservers

Since we began this study in June 2000 we have reported our findings to the root/gTLD server operators, and received much useful feedback in return. The following subsections summarize some of the changes we have observed.

Filtering Bad Queries at the F Root

Figure 9 plots F root performance on Tuesday, 21 November 2000, showing a drop in response time, from 75 to 30 ms at 2230 (i.e., 1530 local time in Palo Alto). We originally thought this was due to the separation of the gTLD and root zones, but now know [16]

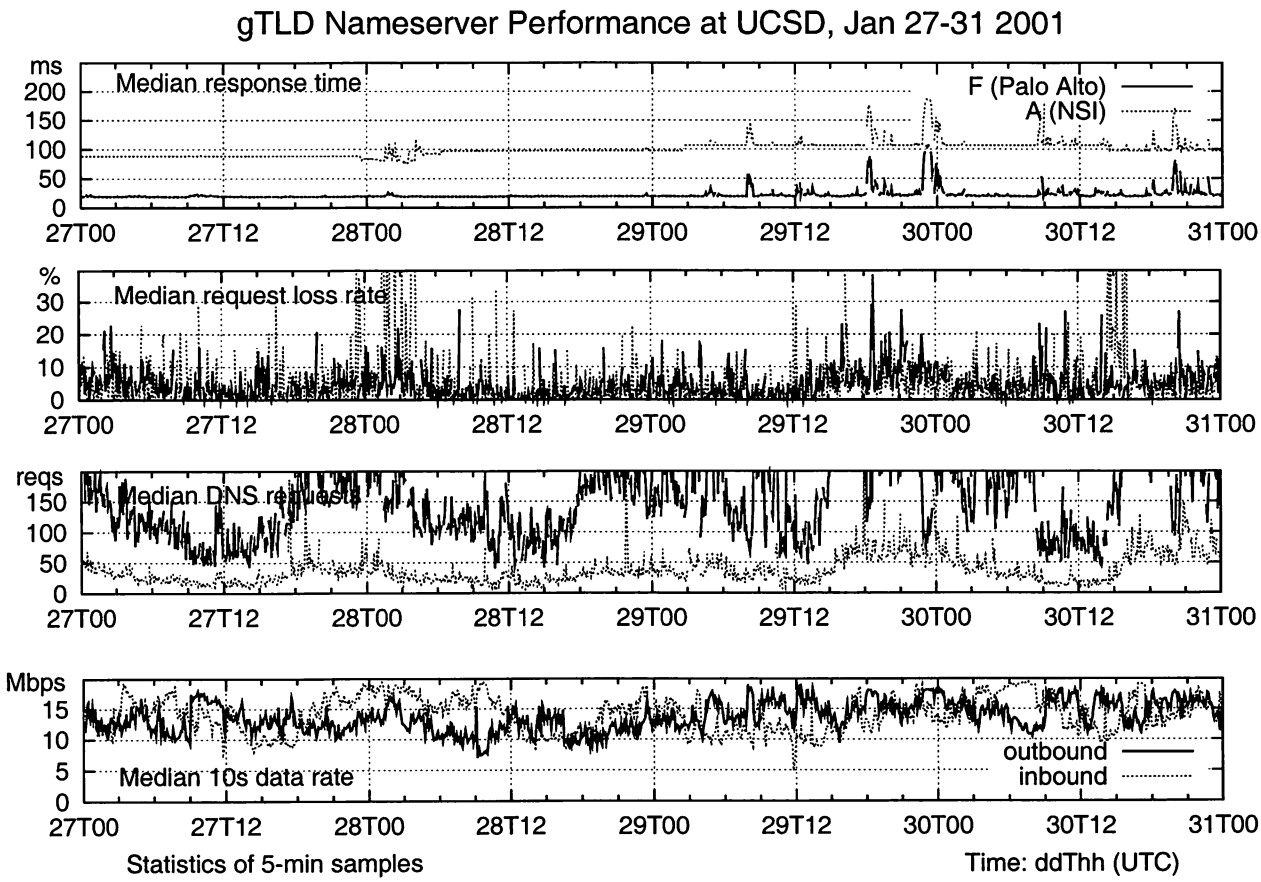


Figure 8: Performance metrics from Figures 5-7 for the F (black lines) and A (grey lines) nameservers. DNS packets for these servers travel via our commodity link. The lower plot shows commodity traffic levels; our commodity link is clearly saturated for long periods. Some of these periods also show spikes in response time or loss rate, but overall there is little correlation between traffic rate, DNS response time, request loss rate or query rate.

that the drop was due to filters introduced by the F root's operator that drop unanswerable packets. The lower response time is accompanied by an increase in the request rate due to BIND's load balancing.

Zone Reloads for the gTLDs

Figure 10 shows 5-minute median loss rates for the gTLD servers in November 2000. We observed a regular daily pattern of 100% loss during a 30 minute period twice a day while *all* the gTLD servers reload the .com zone (>2 GByte). These loss periods correspond to zone transfers, when the machine is reloading the .com zone and is too busy to answer queries.

These prominent loss spikes were also visible in our measurements from New Zealand, in spite of the low DNS request rate there.

Figure 11 shows this behavior in detail for a single day. From this view we can see that the *named.conf* file configuring BIND for these servers allows three zone transfers at a time (parameter `transfers-out = 3`). The servers reload in pairs, the sequence is A+B, C+E+J, D+G+K, and then F+M.

We showed our observations to Network Solutions, who run the gTLD servers, and they changed their reload policy [17]. Each gTLD server is a pair of machines that were both reloading simultaneously. Figure 6 shows that by January 2001, only the C, J, K, and M gTLD servers were still simultaneously reloading, and by the end of July 2001 these zone-reloading-induced losses are completely gone.

Upgrade of the H Root

Performance of the H root server in January 2001 seems abysmal. Indeed, earlier we discussed (Figure 3) the total loss of all H root's DNS requests, and we have also observed H root's poor performance from New Zealand. In January H root had a request loss rate consistently above 90%. Our request rate was low, yet the response time (shown by the occasional dots at about 365 ms on the response time chart, Figure 4) was much higher than for the A root, which is also located on the East Coast of the US. This data suggests that either the network near H or the H server itself was badly under-provisioned.

In our plots for July-August 2001 H is one of the best performing root servers. The story behind H's miraculous recovery is a new system administrator who noticed the very low query rate and load average, and insisted on a hardware upgrade.

In his words [18] (quoted with permission): "Until about June 11th of this year the system was

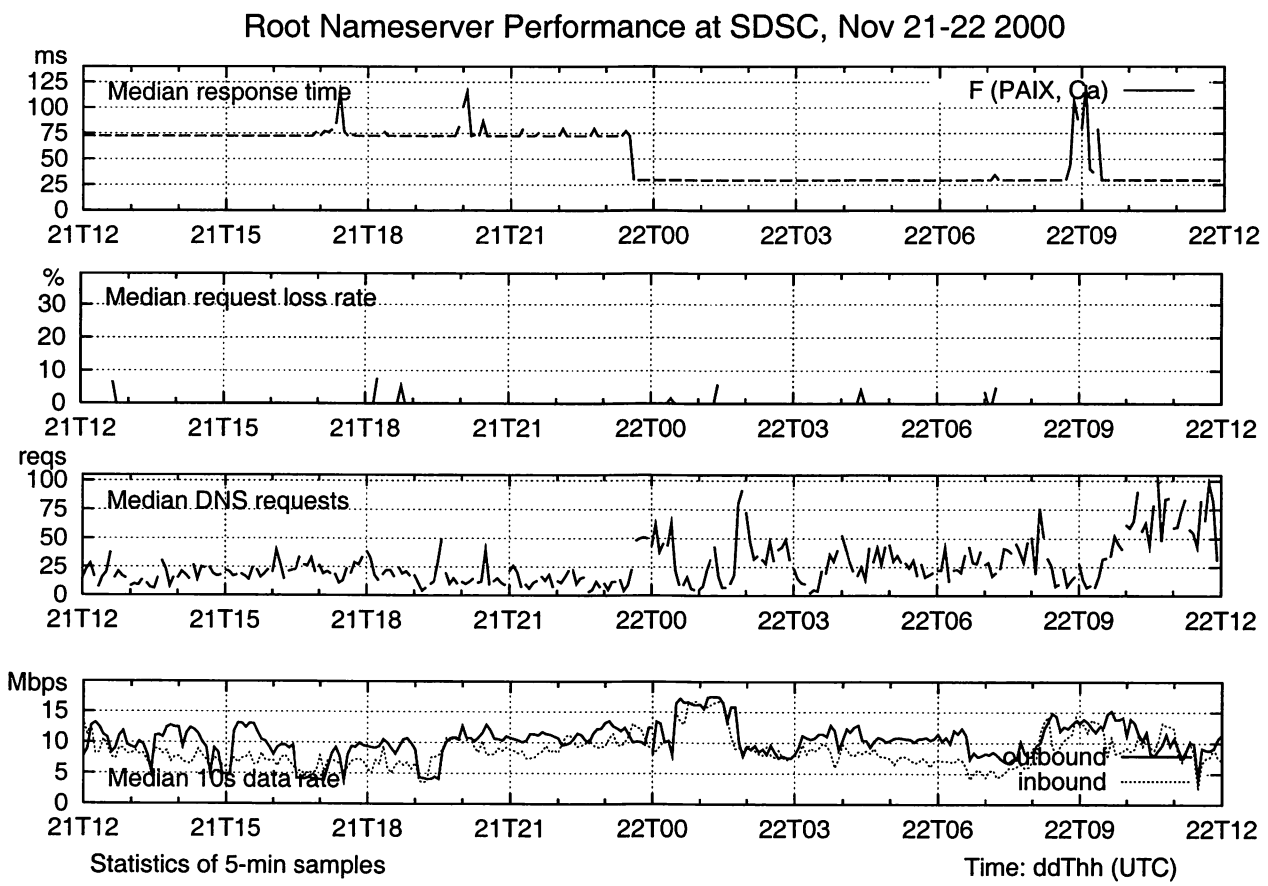


Figure 9: Filtering for unanswerable root queries was introduced on F root at about 2230, Tue 21 Nov. Server response time dropped, and our local BIND responded by increasing its request rate to F.

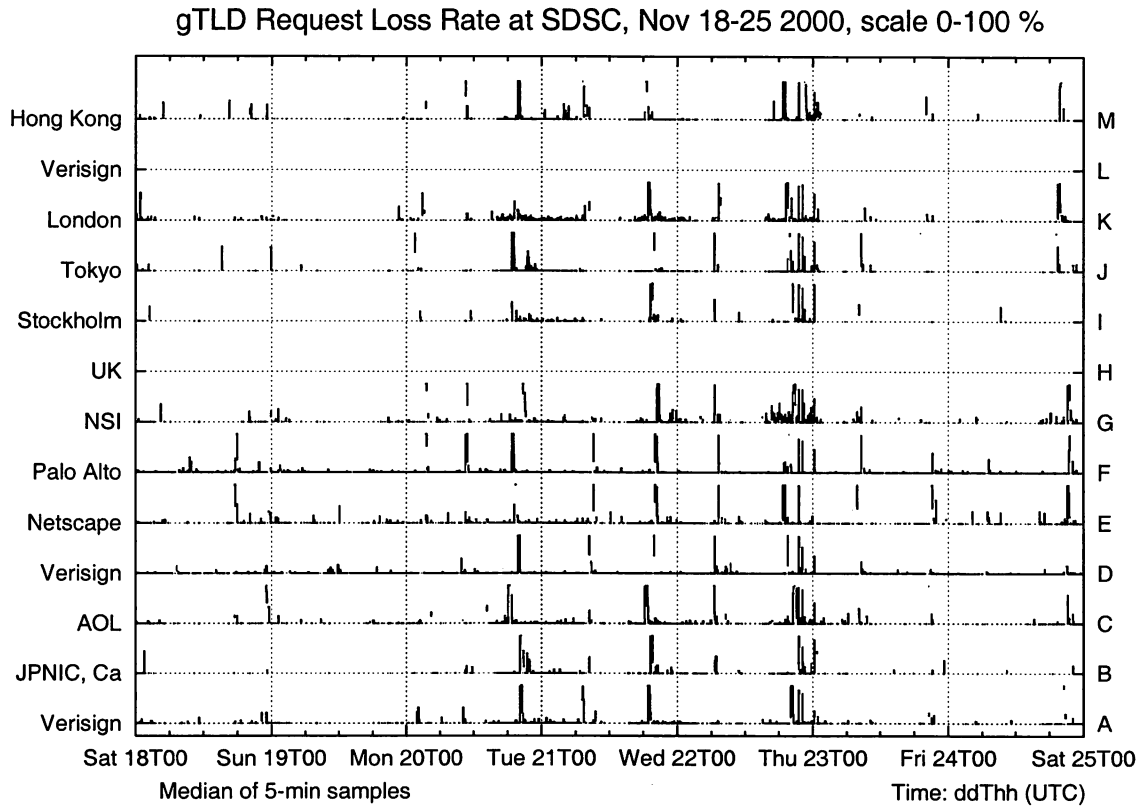


Figure 10: gTLD request loss rate in November 2000, showing high loss (i.e., periods when several servers were not responding) during zone reloads for *all* servers.

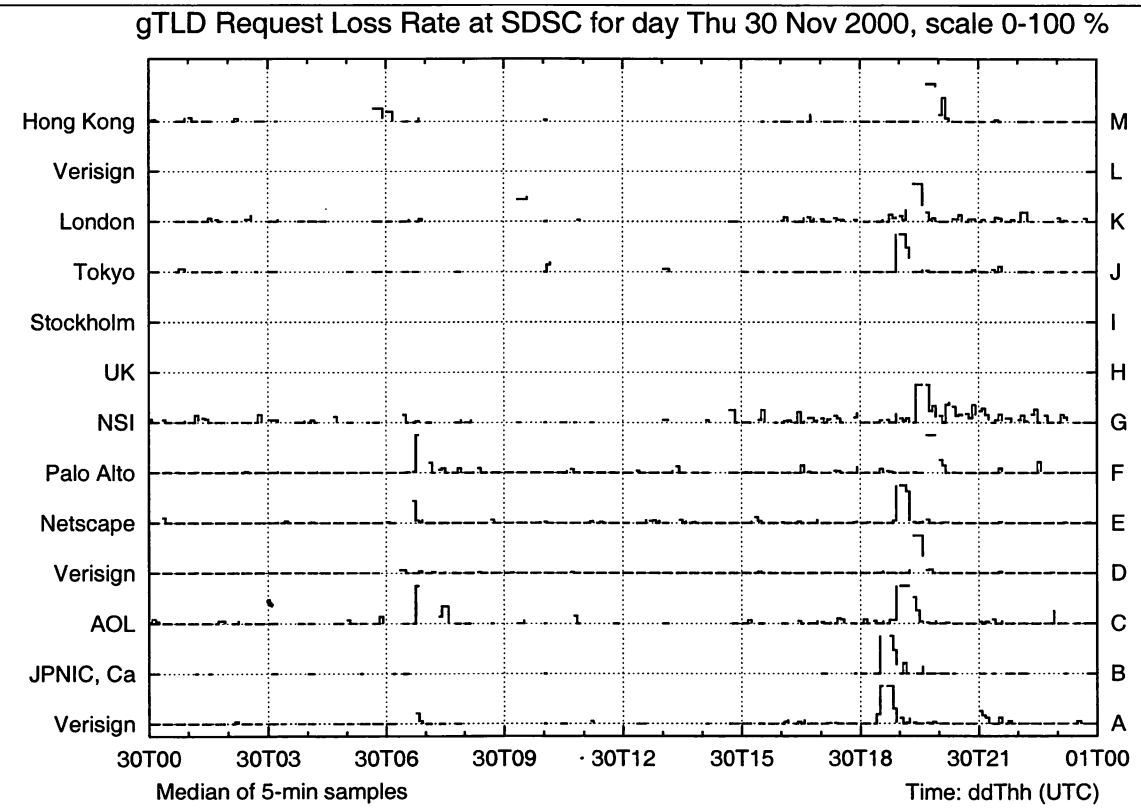


Figure 11: Details of zone reload loss pattern on Thu 30 Nov, showing periods when three servers at a time were inaccessible.

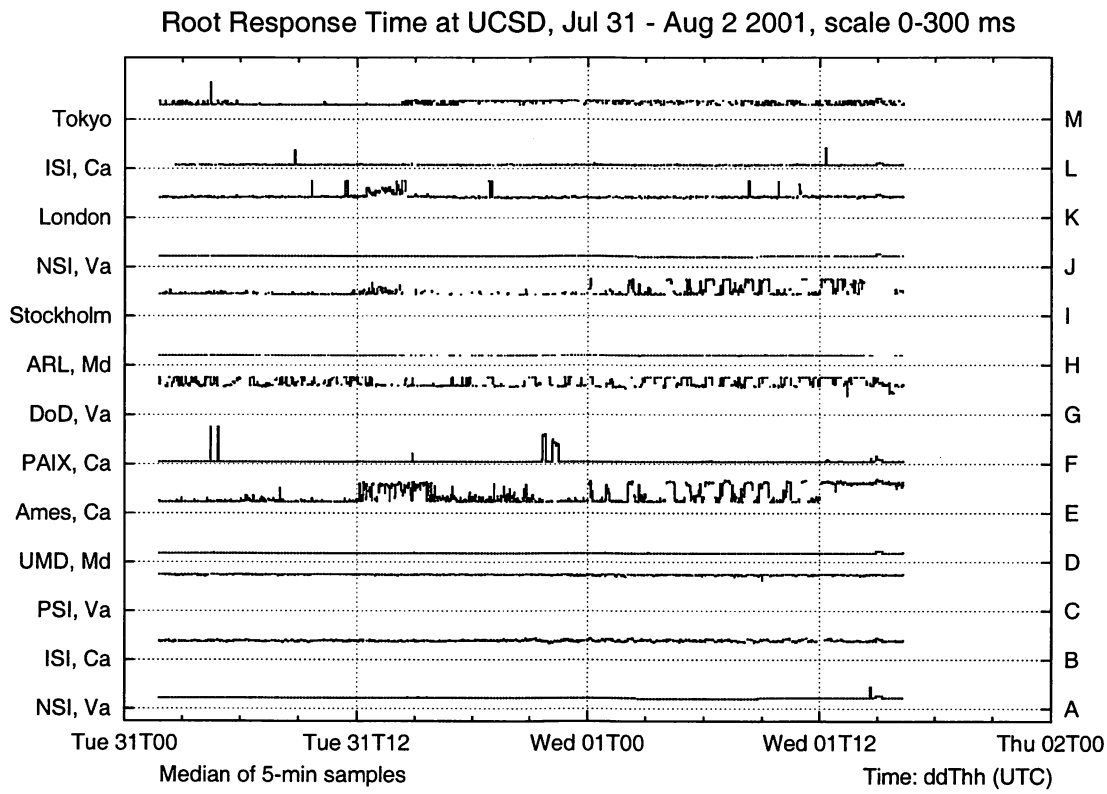


Figure 12: Root server response time in late July 2001. H response is vastly improved, A, B, J and L responses are less variable compared to their response time in January. E and I response times, however, have become more variable.

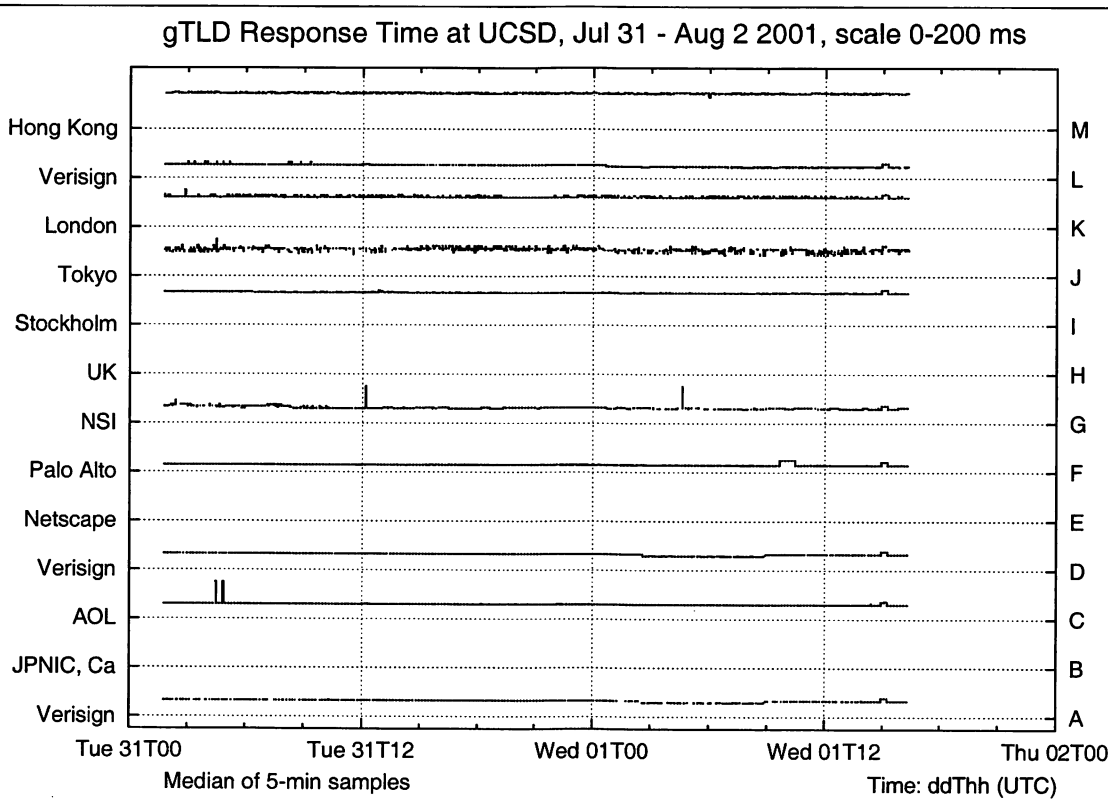


Figure 13: gTLD server response time in late July 2001. J response times remain unchanged, but all the other servers show less variable response times.

running on a 168 MHz Sun Ultra-2. I contend that this, alone, was the reason for the poor performance. After taking over 'H' from the previous administrator, I noticed that 'H' was only able to receive and respond to about 1800 queries/second even though thousands of more queries were being sent to it. Around June 11th, the system was replaced with a >1.2 GHz Intel system and is now 'seeing' between 4000 and 7000 queries per second on average and responding to every one. I've even seen peaks of up to 11,000 queries per second with an equal response rate. I've been collecting statistics since the new server came online if you are interested (10-minute averages). Based on these, the maximum 10-minute average query rate has been 7332.43 queries/second with an average 10-minute query rate of 4259.48 queries/second (8079 data points).

As for H's connectivity, we have a OC12 connection to the Defense Research and Engineering Network (DREN): http://www.hpcmo.hpc.mil/Htdocs/DREN/index.html.

This network has numerous peering locations with the global Internet at such locations as SprintNAP, MAE-East, MAE-West, Fix-West, Seattle GigPoP, Chicago NAP, and transit service through AT&T WorldNet. All are OC3 connections."

Long-term Observations on the Roots and gTLDs

Figure 12 shows 5-minute median response times to the root servers in late July 2001. Our *commodity* meter could not see DNS traffic to the B, E, H or I roots, hence they only appear in our 2001 plots. We have data for the other root severs for 2000 and 2001.

H root now has steady, low response times; this is the most dramatic performance improvement we have observed. Otherwise, the long-term average of the root servers 5-minute median response times have not changed significantly since we began measuring them. We also observe a decrease in the variability of response times, especially for the B root, which was highly variable in 2000, but is steady in 2001. The A, J and L roots had variable response times through January 2001, but these are now (i.e., in July 2001) quite stable. On the other hand, some root servers have become more variable during 2001, for example E and I.

Measured request loss rates have decreased overall. Some of this decrease is due to improvements in our experimental technique, in particular the deployment of our *edu* meter, which greatly reduces the likelihood of missing data due to asymmetric routes from the meter to/from the root and gTLD servers. Another contributing factor is the gTLD servers, which became became operational during the third quarter of 2000, reducing the query load on the root servers, and the traffic congestion on paths to them. In addition, there have also been improvements both in the root servers Internet connectivity and in their capacity to cope with offered load of DNS queries.

Figure 13 shows the gTLD servers 5-minute median response times in July 2001. The J server's response time remains rather variable, i.e., it has not changed since November 2000. However, the degree of variability in response time for the other gTLDs has decreased since November 2000, to the point where their response times are now steady. In November 2000 and January 2001 we saw spikes in response time that were common to all gTLDs located in the US. This effect is no longer visible in July 2001.

In summary, the root and gTLD operators have made considerable efforts to improve the performance of their servers, resulting in worthwhile and observable improvements in the stability of both root and gTLD servers over the last year.

The gTLD servers are well-provisioned, using a single hardware and software architecture. They are well-run, by a single operator. The roots are also well-run, by different operators, using different (sometimes older) hardware and software architectures. This diversity makes them less prone to failures common to a single architecture; we believe that in the long term this hardware and operating system diversity is a desirable feature.

Conclusions and Future Work

Our passive traffic meters have provided an effective method to monitor performance of global nameservers as seen from a client site perspective. Once a meter is configured, 5-minute distributions of response times, together with counts of requests and request loss rates, provide ample data for monitoring performance of the global nameservers and our Internet links to them.

The 'total requests' charts (e.g., Figures 2 and 5) reflect local user activity. They also show which roots and gTLDs are most used by local nameservers; sudden changes indicate a loss of connectivity to one or more global nameservers.

The request loss rate charts (e.g., Figures 3 and 6) show which servers are experiencing short-term congestion or connectivity/routing problems. Problems affecting Internet links close to the measurement site show correlated changes for many of the global nameservers, while problems affecting more distant links appear only on the charts for single servers.

The 'response time' charts (e.g., Figures 4 and 7) indicate how long it takes the root/gTLD nameservers to resolve a DNS query. This metric is important since the delay is often directly visible to, and detrimental to performance for, end users.

The efficiency of the local caching component of the DNS architecture, together with system administrators setting DNS TTLs of the order of minutes to days (rather than seconds to minutes), greatly improves user-perceived delay, since fewer requests

need global lookups. Overall, BIND's load balancing and caching on local nameservers, makes the global DNS extremely reliable (by impressive design).

The DNS 'server performance' plots provide some insight into the way response time and request rate vary with the load on our commodity Internet link. The request loss rate did not, however, show any obvious correlation to other metrics. More work is needed to understand these relationships.

These client-side measurements show that most root servers, as observed from our university, have reasonably low response times, but only fair request loss rates, perhaps due to the rate-limiting on our commodity Internet link. Some servers (C and H for example, and to a lesser degree G) had consistently high loss or high latency or both. Hardware performance may be a contributing factor. Considerable performance improvements have occurred this year, but further investigation of the client-side root/gTLD server measurements is necessary to enable a longer term view of systemic infrastructural performance issues.

Our strip charts are useful for day-to-day monitoring of our Internet connectivity, since they reveal changes in network behavior on paths between our local network and the global servers without our having to send test packets. We are now developing a monitoring tool that will produce these charts in near real time.

A NeTraMet meter, using either live network data, or *tcpdump* trace files, is a useful network administration tool in several ways, including:

- **Monitoring remote DNS servers**, as detailed in this paper. Bear in mind that as well as monitoring the global servers, our strip charts provide a clear indication that a site's local nameservers are load balancing properly.
- **Monitoring local servers**, for example Web servers. As well as measuring request loads and server response times, one might monitor the load balancing between several servers attached to a common network segment.
- **Monitoring traffic on external links**. This traditional use for NeTraMet, i.e., traffic flow measurement, is useful for detecting link saturation, capacity planning, accounting and billing, etc.

NeTraMet is distributed as open-source (GNU Public License) software. It may be downloaded, together with its documentation, from the NeTraMet web site [3]. An introduction to NeTraMet, with discussions on how to configure and use it, is given in [6].

Acknowledgements

We gratefully acknowledge the generous help of our colleagues in implementing the CoralReef version of NeTraMet, in setting up our commodity meter and (at very short notice) our *edu* meter, and in producing diagrams for this paper. Thanks to Hans Werner Braun (NLANR), Bud Hale (NLANR), Bradley Huffaker (CAIDA), Ken Keys (CAIDA), Joerg Micheel (The University of Waikato), David Moore (CAIDA) and Brendan White (CAIDA).

Author Information

Nevil Brownlee co-chaired the IETF's Realtime Traffic Flow Measurement (RTFM) Working Group. He created NeTraMet, an open-source implementation of the RTFM (Internet Standard) architecture at The University of Auckland late in 1992. Nevil now works half time in Auckland overseeing technology developments, particularly those relating to networks, and teaching in Computer Science. He spends the other half of his time at CAIDA in San Diego, where he pursues research into the behavior of Internet traffic, and continues to develop NeTraMet so that it can handle higher-speed networks. Nevil can be reached at nevil@caida.org.

kc claffy is principal investigator for CAIDA, the Cooperative Association for Internet Data Analysis, based at the University of California's San Diego Supercomputer Centre. kc's research interests include: data collection, analysis, and visualization of Internet workload, performance, topology, and routing behavior. She also works on engineering and traffic analysis requirements of the commercial Internet community, often requiring ISP cooperation in the face of commercialization/competition. kc can be reached at kc@caida.org.

Evi Nemeth has been a computer science faculty member at the University of Colorado for years teaching data structures, networking and system administration. In 1998 she visited CAIDA at the University of California, San Diego, where she led the IEC (Internet Engineering Curriculum) effort, and continues to contribute to various CAIDA research activities. Evi is a co-author of the UNIX System Administration Handbook, now in its third edition. She is now moving out of the UNIX and networking worlds and onto her 40 foot sailboat to start exploring the real world. Evi can be reached at evi@caida.org.

References

[1] P. Mockapetris, *Domain Names – Concepts and Facilities*, Internet Standard 0013 (RFCs 1034, 1035), November, 1987.

[2] BIND website, http://www.isc.org/products/BIND/.

[3] NeTraMet website, http://www.auckland.ac.nz/net/NeTraMet/.

[4] CoralReef website, http://www.caida.org/tools/measurement/coralreef.

[5] C. Huitema and S. Weerhandi, *Internet Measurements: the Rising Tide and the DNS Snag*, Monterey ITC Workshop, September, 2000.

[6] N. Brownlee, *Using NeTraMet for Production Traffic Measurement*, Intelligent Management Conference (IM2001), May, 2001.

[7] N. Brownlee and M. Murray, *Streams, Flows and Torrents*, PAM2001 Workshop, April, 2001.

[8] N. Brownlee, kc. claffy, M. Murray and E. Nemeth, *Methodology for Passive Analysis of a University Internet Link*, PAM2001 Workshop, April, 2001.

[9] N. Brownlee, C. Mills and G. Ruth, *Traffic Flow Measurement: Architecture*, RFC 2722, October, 1999.

[10] N. Brownlee, *SRL: A Language for Describing Traffic Flows and Specifying Actions for Flow Groups*, RFC 2723, October, 1999.

[11] S. Handelman, S. Stibler, G. Ruth, *RTFM: New Attributes for Traffic Flow Measurement*, RFC 2724, October, 1999.

[12] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot and E. Lear, *Address Allocation for Private Internets*, RFC 1918, February, 1996.

[13] N. Brownlee, kc. claffy and E. Nemeth, *DNS Measurements at a Root Server*, Globecom 2001, November, 2001.

[14] M. Kuhn, *A Summary of the International Standard Date and Time Notation*, http://www.cl.cam.ac.uk/mgk25/iso-time.html.

[15] CAIDA Topology Mapping website, http://www.caida.org/tools/measurement/skitter/.

[16] Private Communication, Paul Vixie, September, 2001.

[17] Discussion with NSI staff, Wed 13 Dec, 2000.

[18] Private Communication, Howard Kash, August 2001.

Pelican DHCP Automated Self-Registration System: Distributed Registration and Centralized Management

Robin Garner – Tufts University Network Operations

ABSTRACT

Pelican is an automated DHCP client self-registration tool. Unlike other popular self-registration tools, the Pelican registration system runs on an independent server with no access to the DHCP daemon's lease data or configuration files. It derives MAC addresses by SNMP-querying network devices and periodically generating and pushing updated configuration files to all participating DHCP servers. This allows Pelican to scale more effectively than most other existing automated registration systems. Pelican also tracks lease allocation on participating DHCP servers, providing administrators with a central repository of data for simplified administration and troubleshooting.

Introduction

Like most other higher education institutions, Tufts University is experiencing an explosive proliferation of mobile computing devices. Nomadic computing, the ability to move devices around and between campuses, is a self-evident requirement: people buy mobile computing devices because they are "mobile" and expect them to work more or less the same way throughout the University.

The Dynamic Host Configuration Protocol, or DHCP [12], is widely used to enable mobile as well as stationary computing. Some institutions enable nomadic computing by allowing anonymous access to their DHCP services but Tufts' IT governing committees have forbidden anonymous access to our networks; all devices must be registered with the DHCP server(s) in order to be granted a lease. We cannot depend on our users to figure out their device's MAC address and type it into a form ("Is that an 'O' or a zero ?"). Nor can we tell them they need to find a tech support person to help them register: our pool of support staff is small relative to the size of our user population and the proliferation of new technologies has put many additional demands on their time. Thus automated self-registration for DHCP service is absolutely necessary.

Given a fluctuating population of between 3,000 and 9,000 users, a network connectivity or registration trouble-report rate as low as 1% on any given day is still a substantial volume of calls. Support staff must have easy, fast access to aggregated DHCP server and registration system data (including robust logging) in order to provide effective and efficient support for self-registration services and nomadic computing.

Our Pelican DHCP registration tool advances the automated self-registration concepts pioneered by utilities like NetReg [1] and AutoReg [2] by decoupling the registration process from the operation of the DHCP server. By functioning independently, the Pelican server can coalesce all registration information and lease allocation data from many participating DHCP servers into a central repository. This repository can then be used to dynamically generate and distribute configuration files back to those servers. This enables complex correlation and reporting facilities that aid in troubleshooting and accounting.

Related Work

Before creating Pelican, we experimented with many other similar utilities. In the commercial sphere, we evaluated Cisco's Network Registrar [3] and Lucent's QIP [4] products. Both aim to provide comprehensive, integrated DHCP and DNS services with advanced administrative capabilities and their target markets appear to be large corporate installation and ISP's. In 1998 when Pelican was being designed, neither supported automated, externally authenticated self-registration and both were nefariously complicated to operate. The lack of registration facilities severely prejudiced us against both products long before we got around to contemplating pricing, supported architectures, and migration issues.

The non-commercial options that we considered included Southwestern's NetReg [1] and R.I.T.'s AutoReg [2]. These utilities combine automatic self-registration and DHCP system monitoring functionality. The target market for both is institutions of higher-education with large residential populations. Both are based on the same fundamental design concepts:

- Named wildcarding.
- Discrimination between known and unknown clients by the DHCP server and use of address

pools to distinguish between registered and unregistered devices.
- Co-location of the registration agent and the DHCP server on the same physical system.
- Direct parsing and manipulation of the DHCP configuration and lease files by the auto-registration system.
- Assumption of the universality of web browsers; administrators expect that all registrants have web browsers and use them frequently.
- Use of a web-based registration form that accepts username/password pairs and attempts to authenticate to a remote service.

In our trial deployment of NetReg, we found it to be stable and easy to use. It simplified the move-in and registration process dramatically, with the unintended consequence of putting half the residential support staff out of work.

The three main components of a NetReg architecture are a DHCP daemon, an HTTP daemon serving the NetReg scripts, and a specially configured nameserver called a "fake-root nameserver." A fake-root nameserver is one which resolves all name-resolution queries to a single address – the address of the NetReg system. This has the effect of directing all network service requests placed by unknown clients to NetReg. The DHCP daemon and HTTP daemon must be co-located on a single server while the nameserver can be located anywhere on the network, as shown in Figure 1.

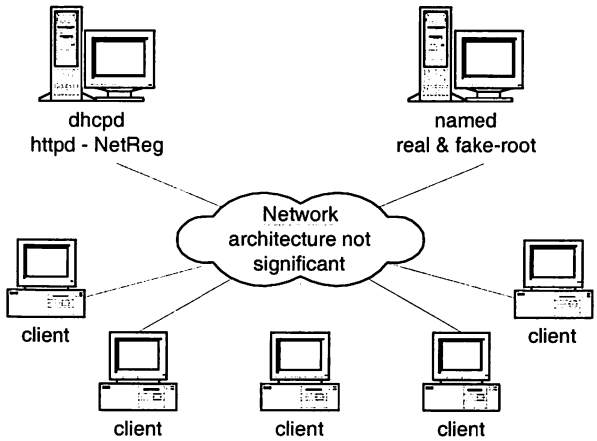


Figure 1: A typical NetReg architecture involves a single DHCP [server, a DNS fake-root server, an arbitrary network architecture, and clients.

```
## NetReg dhcpd.conf
## clients
host foo { hardware ethernet 00:00:00:aa:aa:aa; }
host bar { hardware ethernet 00:00:00:bb:bb:bb; }
host gub { hardware ethernet 00:00:00:cc:cc:cc; }
## subnets
shared-network music-library {
  # KNOWN clients
  #
  subnet 130.64.7.0 netmask 255.255.255.0 {
    option routers 130.64.7.1;
    pool {
      option domain-name-servers 130.64.5.20;
      max-lease-time 86400;
      default-lease-time 86400;
      range 130.64.7.10 130.64.7.240;
      deny unknown clients;
      }
    }
  # UNKNOWN clients
  subnet 10.0.7.0 netmask 255.255.255.0 {
    option routers 10.0.7.1;
    pool {
    option domain-name-servers 10.0.5.2;
      max-lease-time 120;
      default-lease-time 120;
      range 10.0.7.10 10.0.7.240;
      allow unknown clients;
      }
    }
  } # end music-library
# end dhcpd.conf
```

Figure 2: An example NetReg dhcpd.conf file.

Each subnet declaration in the DHCP daemon's dhcpd.conf contains two different pools of addresses: one for known clients and another for unknown clients. A client is known if there is a "host" entry for its MAC address in the dhcpd.conf file. The pool for unknown clients specifies the fake-root nameserver and a very short lease time (five minutes) while the pool for known clients specifies the normal nameserver and a much longer lease time (24 hours). When the DHCP daemon receives a request from a client, it checks to see if the client is known or unknown and allocates an address and nameserver from the appropriate pool.

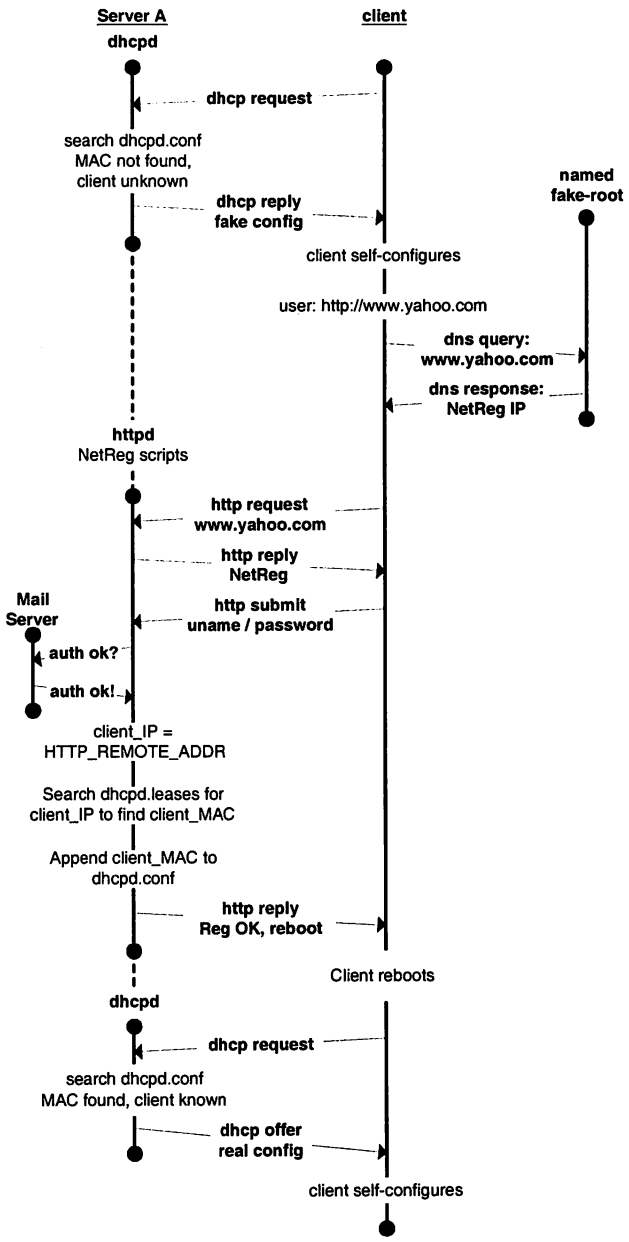


Figure 3: NetReg transaction graph.

In our example configuration in Figure 2, the two pools are drawn from different logical subnets, one a normal publicly routable subnet and the other an RFC1918 private non-routable subnet. Though this is not necessary (both pools can be assigned from a single, routable subnet) we have found that the use of private addresses for unknown clients makes them distinctive and easily identifiable which greatly simplifies troubleshooting. (Supporting two logical subnets on a single LAN requires making a minor configuration change on the routers: a secondary address must be added to each interface, a feature supported by all major vendors.)

A transaction diagram of a typical NetReg registration is shown in Figure 3, with some DHCP-specific steps omitted for clarity. When an unknown client connects, it receives an IP address from the unknown pool, 10.0.7.0/24, and the fake-root nameserver, 10.0.5.2. Any http connections initiated by the client are referred by the fake-root nameserver to the NetReg HTTP daemon, which displays a registration form requesting proof of the user's authenticity. Having verified the user's authentication, NetReg searches the DHCP daemon's dhcpd.leases file for the client's MAC address (keying on the IP address of the HTTP "REMOTE_IP" variable) and appends it to the dhcpd.conf file as a "host" statement. NetReg then restarts the DHCP daemon and displays a web page instructing the user to reboot the client. On reboot, or after five minutes when its lease expires, the client will request a new address from the DHCP daemon and will be granted an address from the known pool, 130.64.7.0/24, and the normal nameserver, 130.64.5.20.

NetReg's simplicity, however, severely limits its scalability: it works very well in a single-server environment but has no mechanism for sharing registration information in a multi-server environment. Our DHCP architecture spans three campuses and involves six servers with overlapping service zones. If we deployed NetReg, we would find that as our users wandered between service zones, they would have to register at each new server to which they connected. Users and their support personnel would find this confusing and inconvenient, particularly given the stunning regularity with which our users forget their passwords. Tracking lease allocation for individual clients would require querying each server individually and collating responses into a useful format would be tricky. These limitations are inherent to the design decision that depends on direct access to the DHCP server's files; they apply not just to NetReg but to all utilities that function in a similar fashion.

Design Specifications

Based in part on our experiences with NetReg and QIP, we formulated the following criteria for a successful automated self-registration system:

- Support for multiple, potentially independent, external authorization and authentication mechanisms.

- Support for more than one DHCP server.
- Maximum ease-of-use: user should need only a web browser, a login ID, and a password on one of the University's authentication systems.
- Automatic prompting to register un-registered devices. No education or tech support intervention necessary.
- Web-based administration interface.
- Alternative supervised registration mechanism ("proxy registration") to support users, visitors, etc. who may not be included in the known authorization scheme or who are unable to authenticate for some reason.
- Support for client data collection that includes user affiliation (for trend analysis, cost accounting, selective culling, etc.), registration date, and expiration date.
- Easily customizable open-source implementation.
- Support for centralized DHCP server lease data collection and aggregation.
- Support for automatic culling of registrations after a configurable interval to prevent the accumulation of stale data without making the user responsible for "de-registering" devices (which would never happen).
- Support for DHCP-style class designations.
- Robust logging.

We analyzed our environment to determine the capacity we would need to accommodate:

- Between 3,000 and 9,000 active registrations at any time during the year.
- An annual influx of roughly 4,500 new registrations over the course of ten to fourteen days each fall.
- Close to a million individual leases, assuming 24-hour lease allocations and a two week data retention rate.

Achieving Scalability: Discovering MAC Addresses

Pelican achieves a degree of scalability impossible in a NetReg-style registration system by decoupling the registration process from the operation of the DHCP system. This allows Pelican to operate in an environment that may include multiple DHCP servers, multiple registration servers, DHCP fail-over, and load-balancing so that, in principle, Pelican can accommodate any size of network and an arbitrarily large client base (Figure 4).

Though a DHCP server's lease file is the most convenient repository of IP to MAC address associations, it is not the only one available. All network devices maintain IP-MAC associations in their ARP tables. Given an IP address, industry-standard SNMP queries to network devices can be used to derive the MAC address associated with the IP. Tufts' network employs routers from three major vendors, all of which support and respond to these queries the same way.

Because the SNMP queries are subnet-based, Pelican must maintain the network address and mask (130.64.7.0/24) of each subnet on which it operates in a database table. This allows Pelican to operate on non-classful subnets like /25's or /18's. Because registration on any particular subnet can be restricted based on user class, the minimum and maximum addresses of each unknown pool are also stored in the database, along with the types of users, if any, that are allowed to register on that subnet. This information is manually entered via web forms.

Pelican does *not* need to know about the network topology; it derives topological information dynamically based on responses to SNMP queries. Pelican must only be given the IP address of one SNMP-enabled router (the "root router") which has routing information for the rest of the network and all routers must be able to respond to SNMP queries from the Pelican server.

To resolve a client's IP address to a MAC address, Pelican determines the client's subnet by comparing its IP address to the Pelican database. It then SNMP-queries the root router and asks if the router has no route, an indirect route, or a direct route to the subnet. A "no route" reply generates an error log message and indicates either a network service interruption or nefarious behavior. An "indirect" response causes Pelican to request the next-hop address on the route. In the manner of traceroute, Pelican follows next-hop addresses through routers until it finds a router that reports back as having a "direct" route, or until it exceeds a configurable maximum number of hops. Pelican queries the router for the interface associated with the route and then for the MAC address associated with the IP address in the interface's ARP cache.

Pelican Components

Implementing Pelican required first making several platform decisions about the software environment in which Pelican would execute.

All Pelican development and operation occurs on UNIX systems here at Tufts, though I imagine that there is no reason it could not be ported fairly easily to a Microsoft platform. All our production servers are Suns running Solaris (v.6 through 8, depending on the system). All are meticulously patched and wrapped by a nocturnal red-haired BOFH.

Pelican uses an SQL-based relational database for data storage. We chose to write Pelican in PHP rather than Perl, not for any important technical reasons, but because we just generally prefer PHP. We chose relatively standard servers: Apache's httpd, ISC's dhcpd, and ISC's named. We briefly flirted with an alternate web server but found that Apache was more reliable. Incidental necessary utilities include awk, ssh, snmp, wget, and cron.

Component Configuration

Pelican was designed as a series of different components which can be run on separate physical systems (Figure 4), but Pelican can also operate on a single machine for small scale deployments and budgets.

Pelican, like NetReg, requires a DHCP daemon, an HTTP daemon, and a fake-root nameserver. Unlike NetReg, each of these may run on separate machines. Additionally, Pelican requires an MySQL database which may also run on an independent machine.

In Pelican, known clients are subcategorized into administrator-defined classes. At Tufts, we assign known clients to either the "student" or "staff" classes based on the registrant's entry in the university's LDAP system, but a default class can be specified as a configuration option.

Pelican's dhcpd.conf file is similar to NetReg's in its use of address pools but there are two key differences. The first is that Pelican includes class specifications that instruct Pelican to match clients to classes based on their MAC addresses. The second is that known clients are identified in NetReg with "host"

```
## Pelican dhcpd.conf

## class definitions

class "staff" {
  match hardware;
  }

class "student" {
  match hardware;
  }

## clients

subclass "staff"   1:00:b0:d0:29:3b:4e;
subclass "staff"   1:00:05:02:01:ea:20;
subclass "staff"   1:00:05:02:00:ff:b5;
subclass "student" 1:00:50:da:09:7f:f7;

## subnets

shared-network music-library {

  # KNOWN clients

  subnet 130.64.7.0 netmask 255.255.255.0 {
    option routers 130.64.7.1;
    option domain-name-servers nameserver.university.edu,
    max-lease-time 86400;
    default-lease-time 86400;
    pool {
      range 130.64.7.200 130.64.7.250;
      allow members of "staff";
      allow members of "student";
      }
    }

  # UNKNOWN clients

  subnet 10.0.7.0 netmask 255.255.255.0 {
    option routers 10.0.7.1;
    pool {
      option domain-name-servers regserver.university.edu;
      max-lease-time 300;
      default-lease-time 300;
      range 10.0.7.11 10.0.7.250;
      allow unknown clients;
      deny known clients;
      deny members of "staff";
      deny members of "student";
      }
    }

  } # end music-library

# end dhcpd.conf
```

Figure 5: An example Pelican dhcpd.conf file.

statements, whereas in Pelican they are identified with "subclass" statements (see Figure 5). MAC addresses specified in subclass statements must have a "1:" prepended onto them.

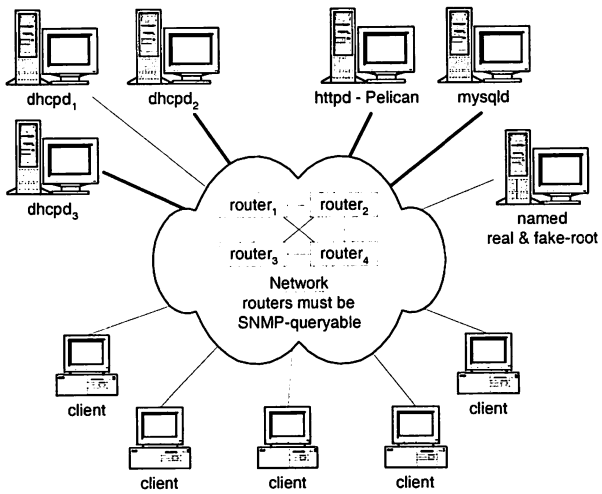


Figure 4: A typical Pelican architecture.

Operation

Pelican has three different and distinct operational processes:

- Client registration.
- Configuration generation and distribution.
- Data collection.

These processes run independently and asynchronously, as shown in Figure 6. A client is properly registered only after processes 1 and 2 have completed one cycle.

Client Registration

Figure 6 illustrates the transactions in a Pelican registration. Unregistered clients making dhcp requests are granted IP addresses from the unknown pool and the fake-root nameserver. The fake-root nameserver resolves all queries to the address of the Pelican server. Clients connecting to the Pelican server are presented with a form inviting them to enter their username and password on any one of three email servers. They are instructed to read the Acceptable Use Policy and indicate their willingness to abide by it by hitting 'Submit' and proceeding with registration.

After verifying that the given username and password correspond to a valid email or shell account, Pelican extracts authorization information from the University LDAP system. The authorization information is used to derive the user's DHCP class ("student" or "staff") and affiliation (department, school, etc.). Pelican confirms that this class of user is permitted to register on the client's subnet by comparing the client's IP address to its dhcp_pool database table. Then the authentication and authorization data is MD5 encrypted and passed back to the client as a cookie

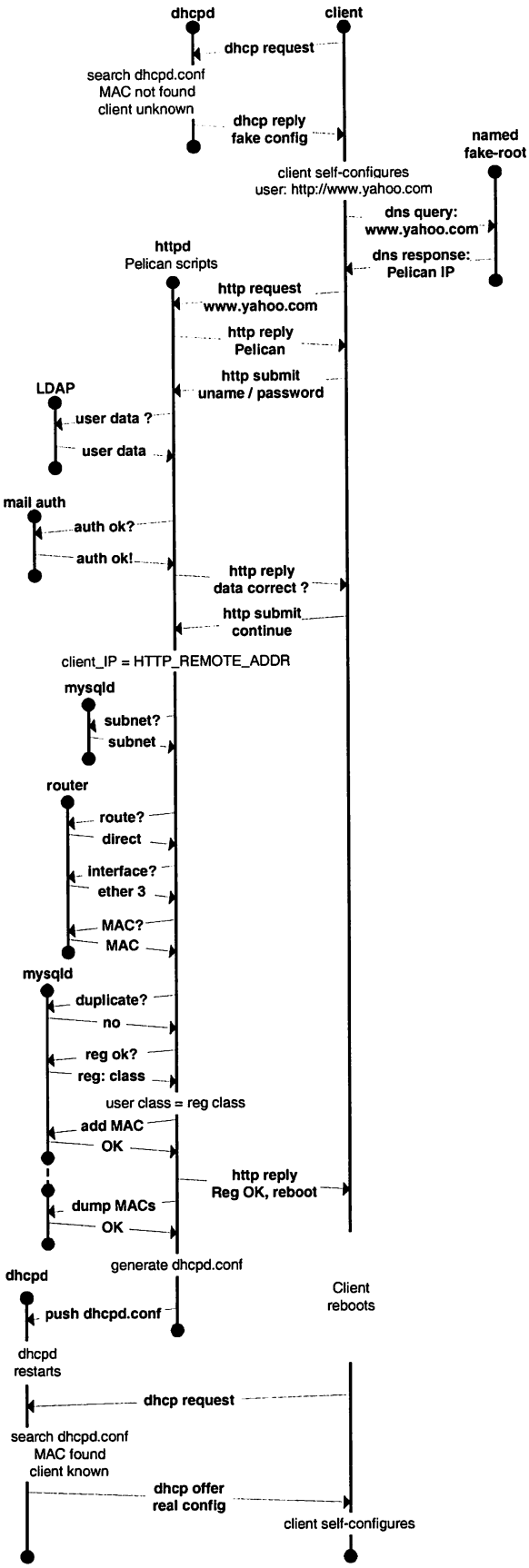


Figure 6: Pelican transactions.

which is used both for session tracking and data storage. The integrity of the cookies is checked at each step of the registration process. The user is presented with a web form displaying the registration data and asking the user to confirm its accuracy by hitting 'submit' to continue the registration process.

Both the authentication and authorization mechanisms are highly configurable: all variables set by the authorization system can be replaced by default values in an environment where an authorization system is either unavailable or considered redundant.

After confirming authorization, authentication, and accuracy, Pelican derives the MAC address of the client and checks the database to ensure that it is not a duplicate registration. The MAC is then inserted into the database along with user's login name, class, and affiliation, and a timestamp and expiration date. The default expiration date is 365 days. Frequently, large areas or groups of clients are registered at the same time (during a cutover, for instance). In order to avoid the inconvenience of having them all expire at the same time, a random number of days between one and 30 is added to each expiration date.

Having successfully added the client, Pelican displays a page asking the user to wait ten minutes and reboot their computer. The client is now registered. Every few minutes, Pelican informs the DHCP servers of all new registrations. On reboot, or after five minutes when its registration lease expires, the client sends an address renewal request to the DHCP server. The DHCP server now recognizes the client as a registered client and declines to renew the original lease. Subsequently, the client requests a new address and is granted a lease for a real, routable address.

In the event that a user is not able to self-register (forgotten password, short-term visitor, etc.), Pelican allows specific users to be designated as having "proxy privileges": the right to register a computer on behalf of someone else. In a proxy registration, the proxy registrant may select an expiration interval: one week, one month, three months, six months, or a year.

Operation

The DHCP servers must periodically be informed of +all new registrations in order to distinguish registered from unregistered clients. Every few minutes (every ten minutes at Tufts), a script is run from cron that dumps the Pelican database into an appropriately formatted DHCP class list. The class list is combined with one or more DHCP server-specific dhcpd.conf files and distributed via ssh to participating DHCP servers. The servers are subsequently restarted.

Data Collection

Pelican collates lease information from many different DHCP servers into its database to allow simple centralized searching. DHCP lease file format is not inherently suitable to be read into a database, so we wrote an awk script that reads the lease file and converts each lease entry into a single line of comma-delimited fields that include the FQDN of the dhcp server, the MAC address of the client, start time, end time, and the uid and name that the client supplied when requesting the lease. This script is installed on each DHCP server, along with an instance of Apache's httpd. The httpd is secured and configured to only accept connections from the Pelican server and only allow access to the lease-parsing script. Every few minutes (again, every ten minutes at Tufts), the Pelican server's crond runs a script that uses wget to connect to the httpd on the DHCP server, activate the awk script, and retrieve its output. The leases are read into the database and can then be easily searched and correlated to individual MAC addresses.

Early in development we used Perl to parse the lease file, but found that awk was substantially faster.

Miscellaneous

Automated client expiration prevents the accumulation of stale data in the database. Every week a cron-initiated script is run which deletes any client whose registration has expired.

A web-based administrative interface allows privileged users to view and search the contents of the Pelican database.

Our DHCP servers are configured to grant "real" address leases for 24 hours and "registration" leases for five minutes. Once the leases have been gathered from the DHCP servers into the database, entries for real addresses are maintained for two weeks while entries for registration addresses are purged after only three days. The purge script is run from cron every night.

Performance

Pelican is ubiquitous at Tufts. It was designed to co-exist with other DHCP infrastructures in order to avoid the necessity of a forklift deployment. We were able to deploy Pelican on a per-subnet or per-building basis, depending on the preferences of front-line support staff.

Deployments

Most of the deployments occurred during the summer or winter breaks when the user population was lowest. Generally, this limited the potential number of registrants to a maximum of 500 at any given time On average, each cutover brought in between 50 and 120 registrants. The vast majority of registrations occurred between 9:00 am and 11:00 am. The highest single-day volume of registrations was 334, with the highest single-hour volume totaling 80 clients (10:00 am to 11:00 am, Jan 12). These numbers are more reflective of human behavior than they are of system performance: registration occurred at the users' time

and pace. Determining the maximum effective registration rate would require scripted testing which we haven't done yet, but we do see multiple registrations per second fairly regularly.

During its first twelve months of operation at Tufts, the entire Pelican apparatus (web server, database, management scripts, user interface) ran on a single multi-purpose Sparc10 with dual 60 Mhz processors. In the middle of student move-in week this past September, load jumped from a daily maximum of 334 registrations to 546. While the Pelican system wasn't directly affected, the DNS fake-root had an unintended consequence: our Sparc10 crashed twice due to the load spike that resulted from the web server trying to service all the automated http queries to microsoft.com generated by machines that were connected to the network but not yet registered. At one point the httpd logged 11,000 bogus requests in fewer than six hours of operation. This volume was generated by fewer than 500 clients, some of which queried at one-second intervals. We were forced to do an emergency migration, splitting parts of the Pelican system between two different machines and allowing the web service to be handled by an Ultra250 with dual 300 Mhz processors. The database, scripting, and administrative interface remained on the original Sparc10. We have not experienced any further performance issues.

During the interval between August 5th and September 5th, we recorded 4,106 registrations on our main campus: 1,418 staff and 2,684 students. Across the three campuses, there are a total of 8,000 registered clients.

Naturally, Pelican performance is highly dependent on the environment: SNMP performance is affected by the architecture and load of the routers in the network. SQl database transactions are affected by the load and configuration of the server. Our SQL server runs on a multi-purpose dual 60 Mhz processor Sparc10. Our network is a heterogeneous collection of Cisco and Foundry routers, some of which do no traffic shaping/filtering and others of which are well occupied in that respect. Packets-per-second loads on the routers also vary widely. There are several wide-area links running at speeds between 1.54 Mb and 10 Mb, multiple gigabit backbones, and LANs generally operate at 100 Mb. Consequently, SNMP statistics tracking the speed with which MAC addresses can be determined are highly context dependent.

With this in mind, these are the statistics we've collected:

SQL insertion speed:

```
High:   43ms
Low:     5ms
Median:  5ms
Avg:     6ms
```

High insertion lags correspond to times when the insertion overlapped with a cron-initiated database dump.

The number of routers traversed is not as significant an indicator of performance as the cpu utilization on those routers. At Tufts, the longest times for MAC determination arise from registrations that traverse two particular routers that are also heavily occupied with filtering and policy enforcement. The shortest times for MAC determination arise during registrations traversing four other routers that do nothing except move packets.

Average total time of execution for the script is under 150 ms, most of which may be attributed to the MAC determination process.

85% of the errors logged by Pelican are due to browsers not having cookies enabled. Though Pelican notifies users of the problem, it would appear that not all users understand what it means to "enable cookie support" or at least are not able to do so without assistance. We logged 410 instances of disabled cookies during 2891 registrations, most of which were corrected by users after a single notification.

10% of the errors were duplicate registrations: people who tried to register their client twice. This may occur when a user fails to reboot their computer after registering and waiting the recommended interval or because the client fails to relinquish network settings when it receives a new DHCP configuration after registering.

The remaining 5% of errors (109 during three weeks of operation) were an assortment that included expired cookies, access denied by the central directory for administrative reasons, and access denied based on user class.

We have seen no evidence of users attempting to tamper with the authentication and authorization data in the cookies.

On average, with 8,000 operating clients and a 2-week retention, our lease table contains 550,000 entries. Retrieving 3600 records from this table based on a partial-string comparison of an IP address ("172.16.237.%") takes 7.5 seconds on a moderately loaded Sparc10.

Operation and Impact

Getting Apache, MySQL, and PHP compiled properly (with LDAP support if you want to do directory lookups is the most challenging aspect of installing and maintaining Pelican. We note with distress that the process does not appear to be getting easier with time or new package revisions.

Nevertheless, Pelican has been running very reliably at Tufts for thirteen months. Network administrators love it: the number of static address requests has dropped from eight per day to eight per week. Because all lease and client data are stored in database tables, we have the ability to cross-join against our ARP log database to identify stolen IP addresses. The ability for users to trivially move around campus has allowed us

to enable more advanced network services, including wireless installations.

It has been very well received by end-users as well: they no longer need to get support assistance to connect a new computer to the network and they can travel easily in and between campuses.

Front-line support staff have found the transition from static addressing to DHCP and class-based access control challenging. Most have absorbed the technical concepts reasonably quickly, though about 5% continue to have difficulty. Most seem to feel that the additional challenge is mitigated by the reduced burden of address requests. Some resent the loss of control and security now that users can roam without their explicit acquiescence.

Additionally, because the University has a distributed support infrastructure, it can be tricky to figure out who is responsible for providing support to users when they are roaming: does support have geographical constraints? How does a support person from one organization and area of campus become familiar with the network landscape elsewhere in order to assist a roaming user? Who, if anyone, provides support for public ports (where Pelican registration is not supported, but roaming for registered clients is) ? How is support handled in areas where students (who have a separate support infrastructure) and staff commingle ? How do users identify public ports? Though we anticipated some of these questions when we initiated development, others caught us by surprise. Our answers are still being developed. Ultimately, by forcing us to reconsider our support model and social landscape, Pelican's impact on Tufts may go far beyond its technical footprint.

Future Work

One of our design assumptions about Pelican was that we would be interested in only a fairly minimal set of data about each client: its MAC address, to whom and by whom it was registered, when it was registered, and when it would expire. Once deployed, it quickly became clear that transactional information about the client was also interesting: its IP address at the time of registration, when it was re-registered, when its user or responsible party was administratively changed, and when it was deleted. Support for retaining some of this information was quickly glommed onto the existing schema, but a more efficient schema would involved two separate tables: one with the minimal client data necessary to construct a dhcp configuration (MAC address and class), and a second table that stored related information about the client (who registered it, when, etc.). Future work will explore the feasibility of the type of schema as well as the possibility of abstracting the database calls in order to support other database engines.

Also, during the initial development of Pelican the decision was made to use cookies to store some of the data necessary to complete registrations because that required the least time to implement. We believe that moving to server-side session tracking will give us more flexibility and reduce our exposure to security risks by hiding more operational details.

Related future work will tie Pelican in with our ARP monitoring system to allow us to identify miscreants who steal addresses without registering. If we get really fancy, we may attempt to implement SNMP-set statements to automatically disable switch ports or block MAC address with selective filters.

Finally, we hope to see Tufts move to an integrated, directory-based authentication/authorization mechanism sometime this year – maybe even with support for digital certificates. Pelican's authentication and authorization code is modular and will be modified as necessary to adapt to advances in related University systems.

Availability

The Pelican scripts, along with our notes on compilation and installation, are available at http://www.net.tufts.edu .

Acknowledgments

Pelican was produced by the author and Peter Radcliffe with invaluable input from Marc Jimenez and Keith Malvetti. All are members of the Tufts University Network Engineering team. Special thanks to Alva Couch for his editing and advice and to Judi Rennie and her team, who helped us troubleshoot under pressure. Thanks also to Mark Mason for his assistance and patience.

Author Information

Robin Garner has been working in the networking field since 1988 and has been a Network Engineer at Tufts University since 1998. Robin can be reached via email at robin@net.tufts.edu or by U. S. mail at Tufts University; 169 Holland St.; TAB 303; Somerville, MA 02144

References

[1] Valian, Peter & Todd Watson, "NetReg: An Automated DHCP Registration System," *Proceedings LISA XIII*, Usenix Assoc., 1999.

[2] Campbell, Matt, "Automatic DHCP Registration," http://www.rit.edu/~mrcsys/dhcp/dhcp98, 1998.

[3] "Cisco Network Registrar," http://www.cisco.com/warp/public/cc/pd/nemnsw/nerr/index.shtml .

[4] "Lucent QIP," http://www.lucent.com/products/solution/0,,CTID+2011-STID+10016-SOID+767-LOCL+1,00.html

[5] Valian, Peter, "NetReg 1.2," http://www.southwestern.edu/ITS/netreg/ , 1999, 2000.

[6] Graves, Rich, "Review of DHCP Registration Systems," http://www.unet.brandeis.edu/~rcgraves/dhcp.html , 1999.

[7] Dromas, Ralph & Ted Lemon, *The DHCP Handbook: Understanding, Deploying, and Managing Automated Configuration Services*, Macmillan Technical Publishing, ISBN 1-57879-137-6, 1999.

[8] Ablitz, Paul & Cricket Liu, *DNS and BIND, Third Edition*, O'Reilly & Associates, ISBN 1-56592-512-2. 1998.

[9] Langfeldt, Nicolai, *The Concise Guide to DNS and BIND*, Que Corporation, ISBN 0-7897-2273-9, 2001.

[10] Stevens, Richard, *TCP/IP Illustrated, Vol. 1*, Addison-Wesley, ISBN 0-201-63346-9, 1994.

[11] Beck, Robert, "Dealing With Public Ethernet Jacks – Switches, Gateways, and Authentication," *Proceedings LISA XIII*, Usenix Assoc., 1999

[12] "Internet Software Consortium Dynamic Host Configuration Protocol (DHCP)," http://www.isc.org/products/DHCP/.

A Management System for Network-Sharable Locally Installed Software: Merging RPM and the Depot Scheme Under Solaris

R. P. C. Rodgers and Ziying Sherwin
– Lister Hill National Center for Biomedical Communications

ABSTRACT

Efficient management of locally installed software is a recurring central theme of system administration. We report here on an experimental merger of two previously independent systems: Redhat's RPM Package Manager (RPM), an open-source database-driven system developed by a major Linux vendor to manage software on a single host; and, an enhanced version of depot, a well-established set of conventions used to manage software that is installed on a server and shared over a network with multiple (possibly heterogeneous) clients. The combination remedies shortcomings in both systems, but to be fully effective, extensions to RPM are required, particularly to its database system. The results of this study point the way toward a second-generation network-distributed version of RPM.

Introduction

Management of the operating system (OS) software on a computer can be time consuming; at many sites, though, the amount of OS software is dwarfed by the amount of additional, locally installed, software arising from a variety of sources. Such software generally provides the services which justify the very existence of the computing facility. The proper installation and maintenance of software is at the heart of system administration, and has a major influence on the utility, reliability, and security of a facility. The work presented here attempts to merge the complimentary features of two open source software installation and management systems: one, known as depot, is a system designed for managing network-shared software; the other, RPM, is designed to manage software on a single host.

Prior Work

One of the earliest attempts to attack the problem of creating and maintaining a network-shared local software repository was the NIST depot scheme [1]. The depot conventions were widely perceived as too complex for smaller facilities run by non-professional administrators, leading to simplified derivatives such as depot-lite [2] and GNU Stow [3]. Colyer, et al. of the Andrew project at CMU offered extensions to the original NIST scheme, including the notion of a software "collection" [4, 5]. Abbey and colleagues at the Advanced Research Laboratory at the University of Texas, Austin (ARL:UT), created a set of perl scripts, opt_depot, which facilitated use of the depot conventions [6, 7]. Other software management schemes that have been developed include STORE [8], the Application Software Installation Server (ASIS) [9], and the /packages scheme employed at Los Alamos National Laboratory (web-based documents for which have been withdrawn from public access). Most of these publically-available systems were developed on UNIX platforms, and attempted to support the sharing of installed software over multiple systems via filesystem-sharing schemes such as network file system (NFS), where clients might be using hardware and OS software different from that of the server. It is difficult to objectively measure the relative costs and benefits of these different approaches; however, STORE and ASIS are much more complex than depot, and none of these systems has been taken up widely outside of its original site of invention.

Commercially-derived systems exist as well. Sun introduced a software management system (pkgadd, pkgrm, pkginfo, ...) as part of its Solaris 2 operating system, using it to install Solaris itself. Functional components that can be installed independently of other components are carved off into their own named "packages," and a list is maintained of where a package's files are installed. Major Linux vendors have all developed software packaging methods with similar intent. These include: Redhat's RPM Package Manager (RPM) [10], which is also used by the French-based MandrakeSoft [11]; Ximian's Red Carpet [12] (and Redhat's equivalent, up2date client); and Debian's Package Management System [13]. Suse's YaST interface appears to be more concerned with OS installation than ongoing local software installation. Caldera International's Volution [14] product is claimed to manage software and other resources over a network of (multi-vendor) Linux hosts. Of these systems, RPM is likely the most widely used, due to Redhat's substantial share of the Linux market as well as to RPM being available as open source for multiple hardware platforms using different UNIX variants. Numerous open source software applications are distributed as RPM

packages. With the exception of Volution, these systems are concerned with management of software on a single host.

Finally, some software applications are bundled with their own installation systems; an example is the XPInstall system employed by the Mozilla web client.

How RPM Works

RPM packages exist in two forms: *binary*-type packages contain executable code for a specific hardware/OS combination, whereas *source*-type packages contain the original source code used to generate the executable binary files. The RPM binary package format is well-defined and consists of four sections: the *lead* (a largely abandoned file structure now used to identify the package), *signature* (the PGP and MD5 data used to validate/authenticate a package), *header* (tag-demarcated information about the package), and *archive* (the files constituting the package, compressed with GNU gzip). For a binary-type RPM package file, the RPM command `rpm -i` does the following: it checks for the presence of any other required packages (*dependency* checking) and for potential conflicts (the overwriting of existing files, or the installation of the current or older versions of already-installed packages); it performs any required pre-installation commands; it installs the files associated with the current package, attempting to preserve local modifications made to configuration files; it performs any required post-installation commands; and, it logs all of the file locations and other package information into the RPM database, which is based on Berkeley DB [15].

For a source-type RPM package file, the `rpm -i` command does much less: it unbundles the source code files and specification file (see below), putting the latter in the SPECS subdirectory. One then uses the `rpm -ba` command to build both binary- and source-type packages.

Both binary- and source-type RPM packages have to be created manually. This process employs various directories that are created when RPM is installed, named BUILD, RPMS, SOURCES, SPECS, and SRPMS, and proceeds as follows:

1. Create a RPM specification (*spec*) file, containing sections which address various aspects of installing and uninstalling a package. The spec file begins with a mandatory preamble consisting of tag-demarcated fields containing general information about the package and its creator. It then continues with one or more of the following sections, as appropriate (using the formal name, beginning with a percent sign):
   - %prep (script to prepare for building);
   - %build (compile/build the package);
   - %pre (optional script to prepare for installation);
   - %install (copy requisite files into place);
   - %clean (clean the build directory tree);
   - %verifyscript (script to verify correct functioning of the package);
   - %post (optional script to be executed after installation);
   - %preun (optional script to prepare for uninstallation);
   - %postun (optional script to be executed after uninstallation); and,
   - %files (list of files to be installed).

   Spec files can make use of macros, OS-specific conditionals, and division of the package into subpackages that can be treated differently from one another.
2. Place the spec file in the RPM SPECS subdirectory.
3. Place the source code in the RPM SOURCES subdirectory (the %prep section of the spec file instructs RPM how to unpack this source and place it in the RPM BUILD directory; macros are available for dealing with common types of non-RPM source packaging such as gzipped tar files).
4. Execute the `rpm -ba` command. This unpacks the sources and places copies in the RPM BUILD subdirectory (a helpful behavior for software distributions which tamper with their own source during the build procedures, as the original source is left intact), changes permissions as required, builds the system from source, installs the compiled and other files where specified and generates binary- and source-type RPM packages, placing them in the RPMS and SRPMS subdirectories, respectively. This maneuver will generally have to be repeated at least twice, as the builder will want to save effort by generating the file list section of the spec file by making a recursive directory listing of the files installed from an earlier pass.

As packages are built, the BUILDS subdirectory gets cleaned out by RPM, but files and packages accumulate in the SPECS, RPMS, and SRPMS directories. Several locations require manual cleaning: the SOURCES subdirectory, and a temporary directory in which log files accumulate in association with failed `rpm -ba` commands.

RPM makes use of MD5 checksums to validate both entire packages and individual files within a package (prior to and after installation), and to guide the treatment of an application's configuration files. It also (optionally) employs PGP [16] to create and authenticate digital signatures for packages. RPM provides utilities to search the RPM database to recover information about installed packages, and to easily update and remove them.

The behavior of RPM can be tailored by system-wide and user-specific initialization files. RPM is built on the rpmlib library, which has an Application Programmer's Interface comprising over 60 different functions.

How Depot Works

Henceforth in this paper, depot refers to conventions for software management employed at the U. S. National Library of Medicine (NLM), relying upon modified versions of the ARL:UT perl scripts (originally known as opt_depot, and building upon the earlier work at NIST [1] and CMU [4, 5]).

The method employs the following directory tree on a server: */depot_server/<hardware-type>/<OS-type>/package*, which allows the server to provide files for multiple arbitrary hardware/OS combinations. An individual package exists within its own subdirectory within the above path, and is named for the package and its version number (for example, */depot_server/sparc/SunOS5.8/package/gcc_3.0*).

Within such a package directory, individual files must be installed within the following subdirectories (this list is locally configurable): *app-defaults* (X windows app-defaults files), *bin* (binaries), *html* (HTML documentation), *include* (include files), *info* (TeXinfo files), *javaclass* (Java class files), *lib* (library files), *man* (UNIX manual pages), *pdf* (PDF documentation), and *sbin* (administrative binaries); UNIX manual pages are organized within a package's *man* subdirectory in subdirectories (*man1*, *man1m*, *man3*, ...) in accord with System V UNIX manual section numbering conventions. If a package has requirements which preclude following this convention (as for example, with some commercial software), it is installed within a subdirectory named *vendor*, and links are made from files or subdirectories within the *vendor* subdirectory to the appropriate *app-defaults* ... *sbin* subdirectories.

On a depot client, a directory with a name of the form */depot_mount/<server-name>/package* is present, where *<server-name>* represents a particular depot server. One such directory is present for each depot server that is providing software to this client. Package subdirectories from each server's */depot_server/<hardware-type>/<OS-type>/package/* directory are mounted into the client's corresponding */depot_mount/<server-name>/package* directory, using a network file-sharing scheme such as NFS.

The client also has a */depot* directory, which contains subdirectories as listed above for the server package directories (*app-defaults* ... *sbin*, excluding *vendor*). Entire packages from */depot_mount/...* are symbolically linked into */depot/package* (for example, */depot/package/ gcc_3.0*). In addition, the files appearing within a package are linked into the corresponding directories of */depot* (for example, */depot/package/gcc_3.0/bin/gcc* is linked to */depot/bin/gcc*). Finally, if a package must write into host-specific files (for example, log or database files), these are placed in the directory */var/depot/<package-name>*

A server (or standalone host) may act as a client to itself, and possess */depot_mount*, */depot*, and */var/depot* directories as well, although we employ symbolic links rather than NFS to provide files to the */depot_mount* tree in that case.

The behavior of depot on a client is controlled by configuration files: */depot/site* controls the mounting of files from multiple depot servers; */depot/.exclude* prevents specified packages from being linked into */depot/package*; and, */depot/.priority* controls which packages have priority for linking into */depot/{bin, html, ... sbin}* when there are name conflicts between individual files coming from different packages.

Thus far we have described *shared* depot packages, which a server provides to one or more depot client hosts. Packages that are specific to a client (for example, node-locked commercial products, or software that requires hardware that is specific to the client) can be installed directly into */depot/package* as a *local* package; its files are linked into */depot/{bin, html, ..., sbin}* along with the shared packages, subject to the same configuration files.

Although this description may make depot seem complicated, in practice it is not. The main labor is learning how to make a new software package conform to depot's package directory structuring conventions. We often employ script wrappers to encapsulate actual binaries, which allows us to set up various environment variables for a given application, freeing the user from having to do so. The perl scripts of the ARL:UT depot system automate maintenance of the underlying system of links.

Shared and Complementary Features of Depot and RPM

Both RPM and depot provide structured, disciplined means of managing software installations. Not surprisingly, they address many issues in common, but with varying degrees of rigor:

1. Both RPM binary-type and installed depot packages can be easily installed on additional hosts. In the case of RPM, this is done by copying the binary-type package to the target host and using the rpm -i command; in the case of depot, this is done by copying the directory for the installed package from */depot_server/<hardware-type>/<OS-type>/package* (for a shared package), or */depot/package* (for a local package) to the target host (either client or server) and running a depot script to update the requisite symbolic links (cron can be used to automate the latter). Both reduce the number of times that a given piece of software must be installed from scratch: the original installer/packager can better afford to focus upon installing the package correctly, as it must be done only once.
2. Both allow easy uninstallation of packages. With depot, packages can be cleanly removed by a single rm command followed by execution of a depot script to clean up unresolved symbolic links (using cron to automate this if

desired). RPM automates the taking of additional actions through its uninstallation scripts.
3. Both methods attempt to document the installed software. At NLM, each depot package has in its root directory a manually constructed file named README.LOCAL, which includes a header containing defined fields describing the package, its author and origin, and its installer. There is considerable overlap with the information contained in the RPM spec file.
4. Both systems can support multiple versions of a given package, if and only if the package in question does not require the use of absolute file paths; with RPM, this requires use of the `--relocate` flag.

The two systems compliment one another in a number of important respects:

1. Applicability over a network. Unlike RPM, depot is inherently designed to accommodate use of a network. Multiple depot servers can provide redundancy (through NFS roll-over). Collaboration is facilitated, as different workgroups can specialize in particular types of software on their own depot server, and share the results within their larger organization.
2. Dependency checking. Under NLM depot, the installer places dependency information in the README.LOCAL file that gets installed with the package. This manual process is error-prone. RPM employs the UNIX ldd command to determine which libraries the package requires (its "dependencies"), and logs this information into the RPM database. At installation and uninstallation time, dependencies can be handled rigorously.
3. Package documentation. The README.LOCAL file remains as a human-readable document with the installed package; various components of the RPM spec file become part of the RPM database record, but the spec file does not remain online as part of the installed package. Unlike README.LOCAL, a spec file can contain procedural components (various scripts) that get automatically invoked at the appropriate moment. Unlike the spec file, README.LOCAL includes (manual) instructions for host- and user-specific installation steps that a package may require to become fully functional, reflecting the multi-host networked nature of depot. For example, the GNU findutils system requires that a database be built on each client host, and Sun's StarOffice system requires that each user run an initialization script prior to using the package. README.LOCAL also contains transcripts of the installation procedure and copies of email correspondence with other parties in connection with the software.
4. User environment. Under depot, search paths are short and simple (*/depot/bin*, */depot/man*, ...).

The Experimental Environment

The study was done using an UltraSPARC 2 as the depot server, and an UltraSPARC 2 and two UltraSPARC 60 machines as depot clients. The machines were operating under Solaris 2.[5-8]. We standardized the naming and NFS automounting schemes used by depot as described above. At our request, Abbey and colleagues added new features to the original ARL:UT opt_depot scripts: the ability to mount software packages on a client from multiple depot servers; and, improved configurability of the perl scripts. We installed and used depot routinely for a period of four years, successfully supporting as many as two different versions of Solaris concurrently. At the time of writing, the depot server contained 321 shared packages and 21 local packages for Solaris 2.8. Source for the SPARC-compatible version RPM 4.0.2 was obtained from the web site http://www.rpm.org. Eighteen lines of the code had to be modified to get it to compile under Solaris 2.8 using gcc 3.0. RPM was installed as a shared depot package on the depot server, with the RPM database files placed in */depot/package/rpm_4.0.2/vendor/var/lib/rpm*. Information about all shared packages is logged into this database. On each client, the RPM database is installed in */var/depot/rpm_4.0.2/local_db*, and information about local packages is placed there. The need for and limitations of using two databases is discussed below.

To allow RPM dependency checking to operate, we employed a script, vpkg-provides2.sh, provided with the Solaris version of RPM, which uses the information provided by Sun's proprietary package database to create entries for "virtual" RPM packages (there were 564 such packages on our depot server). We then installed a number of packages using RPM, while following the depot conventions: a library (libpcap 0.4); an application depending upon that library (snort 1.7); a self-standing source application (wget 1.6), and, a commercial pre-built binary application (netscape 4.77).

Results/Discussion

The RPM/depot merger experiment suggested a number of technical directions for future work:

1. The creation of "virtual" RPM packages from Sun's proprietary package format is slow, and the script must be rerun when additional Sun packages are installed. Ideally, Sun should use RPM/depot; otherwise, the script should be improved, and wrappers created for the Sun package tools to invisibly and reliably integrate them with RPM/depot.
2. RPM and depot functionalities should be coupled. RPM spec file macros could be used to invoke requisite depot scripts during installation/uninstallation, and to automate actions now taken manually. Alternatively, the scripts could be invoked directly from RPM source code.

3. Capabilities of the RPM spec and depot README.LOCAL files should be merged; in particular, RPM needs to know how to trigger the host- and user-specific initialization (and uninstallation) steps required by some software. Other helpful aspects of depot as used at NLM: inclusion of installation transcripts and related email correspondence, and leaving a human-readable document with the installed package. Transcripts of the build could be created by using spec file macros to capture the process in a subshell, redirecting the output to a file which is then incorporated into the package documentation. It would be ideal if the README.LOCAL/spec merger resulted in redundancy, such that the README.LOCAL files could be used to rapidly rebuild a corrupted RPM database, and to manually manage packages in an emergency.
4. Modifications to RPM to allow dependency checking over a network. RPM can only use one database at a time; either the default one, or one supplied following the rpm command line argument `--dbpath`. As described earlier, we installed RPM with two databases: one database (on the server) containing the information for shared packages (shared by all hosts over the network), and a second database for local packages (on each client). Most packages can be shared, and use of the shared package database will succeed much of the time, as it contains records for most of the Solaris virtual packages as well as the shared depot packages. Installation of local packages will fail more often, as the local package database will not contain information about shared packages upon which local ones may depend. This problem could be partially solved by modifying RPM, supporting the searching of a comma-separated list of databases instead of just a single database (an enhancement that has already been requested by other RPM users; see bugzilla request #4137 on the Redhat web site). This is preferable to error-prone work-arounds such as cloning the content of the shared database onto the local databases. This solution is partial, however, because of the way in which depot handles multiple versions of packages and file name collisions, through the depot configuration files. RPM should be able to deal with these files, so as to know how to get to the files a new package needs.
5. User-defined tags are not supported for the RPM spec file and database; such tags would be useful for local extensions to the database.
6. Other RPM enhancements could be achieved more efficiently by means of modest code modifications. When creating an installable RPM binary-type package, its installation path must be configured for either a local or shared depot package. RPM could be altered to allow the package to be designated to be installed as shared or local, and alter the installation automatically. (currently, one can use the RPM `--relocate` command-line option, with appropriate path argument, to install a shared package as a local package, or vice versa).
7. One of depot's strengths is its ability to offer files for a variety of operating systems, whereas RPM is UNIX-specific. If RPM were to be made to support non-UNIX target operating systems, it would have to know how to check for dependencies and to build packages on the target OS. This would be a considerable undertaking compared to the changes suggested earlier. However, one could still concurrently employ RPM/depot for enhanced control of UNIX software, while continuing to use depot as at present to support the other OSs.

Conclusion

Automation and standardization are two means of reducing the considerable costs of administering software on multiple hosts. Depot is a highly efficient means of managing local software, even on standalone systems; its benefits are compounded when used with multiple networked hosts, an environment in which one can use both network-shared and local (client-specific) depot packages. RPM is better at certain things such as documenting packages by database, and dependency checking, but is currently designed for use on a single machine. Our experiment in merging RPM and depot is a qualified success, in that most of our software can be installed and used with RPM/depot without modifying RPM or depot. A fully functioning RPM/depot system, however, requires slight modifications to RPM source code: most importantly, to allow the searching of multiple RPM databases to support dependency checking for local depot packages; less importantly, to couple execution of depot scripts to the execution of RPM commands, and to support the automatic reconfiguration of paths required when installing as a local depot package one that was originally designed to be shared (or vice versa).

Such modifications seem a slight price to pay in order to turn RPM into a network-based software management tool. It would also make the system more attractive as a packaging system for use by other UNIX/Linux vendors. The existence of a single widely-shared system could save time for administrators, by allowing the creation of ftp- and Web-accessible archives of RPM/depot packages for Solaris (and other) platforms, greatly reducing installation effort.

Modifications to RPM to support non-UNIX clients would be more complex than the ones just described, and are harder to justify.

Code Availability

The code and documentation for RPM/depot for Solaris will be available at the time of the conference, from: http://www.etg.nlm.nih.gov .

Acknowledgements

We wish to thank Jeff Johnson of Redhat for invaluable assistance with installing and using RPM under Solaris. Jonathan Abbey of ARL:UT assisted us in understanding and using his opt_depot scripts, and made helpful extensions to them at our request. Both provided helpful remarks about the manuscript, along with Jules Aronson of NLM and Nelson Beebe of the University of Utah.

Funding Sources & Copyright

Biographical Notes

R. P. C. Rodgers (rodgers@nlm.nih.gov) works in biomedical informatics at the Lister Hill National Center for Biomedical Communications (LHNCBC), where he heads the Emerging Technologies Group. He received a B.A. From Harvard College in 1972, a M.D. from the University of Utah College of Medicine in 1976, and postdoctoral training from the University of London, University of Louvain, the National Cancer Institute, and the University of California, San Francisco (UCSF). He served on the faculty at UCSF prior to joining LHNCBC, a research arm of the U. S. National Library of Medicine (NLM). At NLM he became an early and active exponent of the World Wide Web, creating and running NLM's web services for the first two years of their existence. He has participated in a number of IETF working groups, and served as a founding member of the International World Wide Web Conference Committee and founding chair of the NSF/NCSA World Wide Web Federal Consortium.

Ziying Sherwin (sherwin@nlm.nih.gov) received a B.S. in Computing & Engineering from Zhejiang University in 1996, and a M.S. in Computer & Information Science from the University of Delaware in 1999. She has worked Bell for Atlantic, and joined the Emerging Technologies Group at LHNCBC in 2000.

References

[1] Manheimer, K., B. Warsaw, S. N. Clark, and W. Rowe, "The Depot: A Framework for Sharing Software Installation Across Organizational and UNIX Platform Boundaries," *LISA IV*, http://www.forwiss.uni-passau.de/archive/marchiv/systemverwaltung.html, 17-19 October, 1990.

[2] Rouillard, J. P., and R. B. Martin, "Depot-Lite: A Mechanism for Managing Software," http://www.usenix.org/publications/library/proceedings/lisa94/martin.html, *LISA VIII,* 1994.

[3] Glickstein, B., "GNU Stow," http://www.gnu.ai.mit.edu/software/stow/ , http://www.gnu.ai.mit.edu/software/stow/manual.html .

[4] Colyer, W., and W. Wong, "Depot: A Tool for Managing Software Environments," *LISA VI,* http://andrew2.andrew.cmu.edu/depot/depot-lisaVI-paper.html, 1992.

[5] "The Depot Configuration Management Project," Carnegie Mellon University, http://andrew2.andrew.cmu.edu/ANDREWII/depot.html, http://asg.web.cmu.edu/depot/depot.html.

[6] "opt_depot," ARL, University of Texas at Austin, http://www.arlut.utexas.edu/csd/opt_depot/opt_depot.html .

[7] Abbey, J., "The Group Administration Shell and the GASH Network Computing Environment," *LISA VIII*, http://www.arlut.utexas.edu/csd/gash_docs/lisa_paper/paper.html, September, 1994.

[8] Bakken, S. S., A. Christensen, T. Egge, and A. H. Juul, "STORE," Norwegian University of Science and Technology, http://www.pvv.unit.no/~arnej/store/storedoc.html .

[9] Defert, P., S. Gouache, A. Peyrat, and I. Reguero, "ASIS User's and Reference Guide, Version 3.95," http://consult.cern.ch/writeups/asis/node1.html, CERN, 1997.

[10] Bailey, E. C., *Maximum RPM,* SAMS Publishing http://www.rpm.org/max-rpm/index.html; http://www.rpmdp.org/rpmbook, 1997.

[11] Bégnis, C., G. Cottenceau, G. Lee, and T. Vignaud, *Mandrake RPM HOWTO, vol. 1.1*, http://www.linux-mandrake.com/en/howtos/mdk-rpm/ .

[12] Ximian, "Red Carpet," http://www.ximian.com/products/ximian_red_carpet/ .

[13] Debian, "Package Management System," http://www.debian.org/doc/FAQ/ch-pkg_basics.html, http://www.debian.org/doc/packaging-manuals/developers-reference/ .

[14] Caldera International, "Volution," http://www.caldera.com/products/volution/ .

[15] Sleepycat Software Inc., *Berkeley DB*, New Riders Publishing, Indianapolis, 2001.

[16] Garfinkel, S., *PGP: Pretty Good Privacy*, First Edition, December, 1994.

File Distribution Efficiencies: cfengine vs. rsync

Andrew Mayhew – Logictier, Inc.

ABSTRACT

This papers reports on a preliminary investigation of the performance of cfengine and rsync for file distribution for the purposes of system maintenance. It focuses on two aspects of file distribution: the transfer during an initial copy from a master server to a client, and the file verification of the client's files against the server's files. It is shown that for larger file transfers rsync performs better, while cfengine manages smaller transfers better.

Introduction

In originally implementing cfengine (v1.5.4) for host management on our network, we ran across several problems with the file transfer protocol [smith]. Due to faults which would halt transfers mid-session we decided that a replacement method for synchronizing applications and configurations between the servers and individual nodes was needed. We did not, however, wish to completely get rid of cfengine, because we still found its other features for system immunization useful. As such, we implemented an rsync over SSH system controlled and run from each particular host's cfengine agent. While this has worked quite well, this kludge introduces some possible instabilities and quirks in operations.

As cfengine has matured, many of the problems and bugs that we originally faced have been addressed. In reviewing our file synchronization method against current versions of cfengine, we decided to more systematically test various copy methods available to see which was really most suitable for our needs. While there are several other file-copying systems available, our testing is focused on cfengine and rsync and how various levels of securing the network affect transfer speeds.

Testing Methodology

The repository server was a Sun E450 (four 440 MHz UltraSparc CPU's and 2 GB of RAM) running Solaris 7. The clients being served in these tests were a Dell PowerEdge 2450 (two 700 MHz PIII CPU's and 512 MB of RAM) running RedHat Linux 6.2, and a Sun Netra T1 (one 440 MHz UltraSparc CPU and 1 GB of RAM) running Solaris 7. In all cases, the machines were running near completely idle with only a minimum number of required processes running in order to obviate any possible interaction issues. All hosts ran 100 mbit full-duplex Ethernet connected to a single switch isolated from the rest of our network, to eliminate any possible networking issues. Each test run was done in a serial fashion so that only one client was communicating with the server at a given time.

The versions of the file transfer applications used at the time were cfengine v1.6.3 and rsync v2.3.1. Cfengine was additionally patched to be able to directly call any external copy method desired by the user [patch]. A typical rsync incantation and similar cfengine configuration are shown in Listing 1 for example purposes.

There were four different copy configurations run with four different-sized transfers. The copy configurations were:

- **Baseline cfengine copy**: This uses the native non-encrypted cfengine copy method.
- **Tunneled cfengine copy**: Native cfengine copy method run over an SSH tunnel.
- **3DES cfengine copy**: Native encrypted cfengine copy method.

```
rsync -qrplogDze "ssh -l cfengine -i .identity -o StrictHostKeyChecking=no" \
                        server:/master/repository /local/destination

copy:
        /master/repository      dest=/local/destination
                mode=0444
                secure=true
                recurse=inf
                server=$(cfd_master)
                ignore=CVS
                backup=false
                type=md5
                purge=false
```

Listing 1: Typical rsync invocation.

- **External rsync copy**: Externally called from cfengine rsync copy with SSH as the rsync transport.

The four different-sized transfers were as follows:

- Small: 128K, 22 files, 1 directory
- Medium: 2096K, 51 files, 2 directories
- Large: 209096K, 4726 files, 454 directories
- Larger: 258388K, 5646 files, 528 directories

The above configurations were run five times on each machine at each transfer size with debugging first turned on and then again with debugging turned off, to produce 800 data points (4 transfer setups * 4 different-sized transfers * 5 runs with debugging on * 5 runs with debugging off * 2 client machines). Between runs, the copied files were removed. The whole test cycle was then repeated to test file verification speeds, for an additional 800 data points [raw]. File verification consisted of the process of subsequent runs to check the files against the server for changes, either locally or on the server. Our tests for verification were performed with no changes to the files on either client or server.

Results

For the actual transfer of data, the various methods ordered themselves out fairly evenly. While rsync

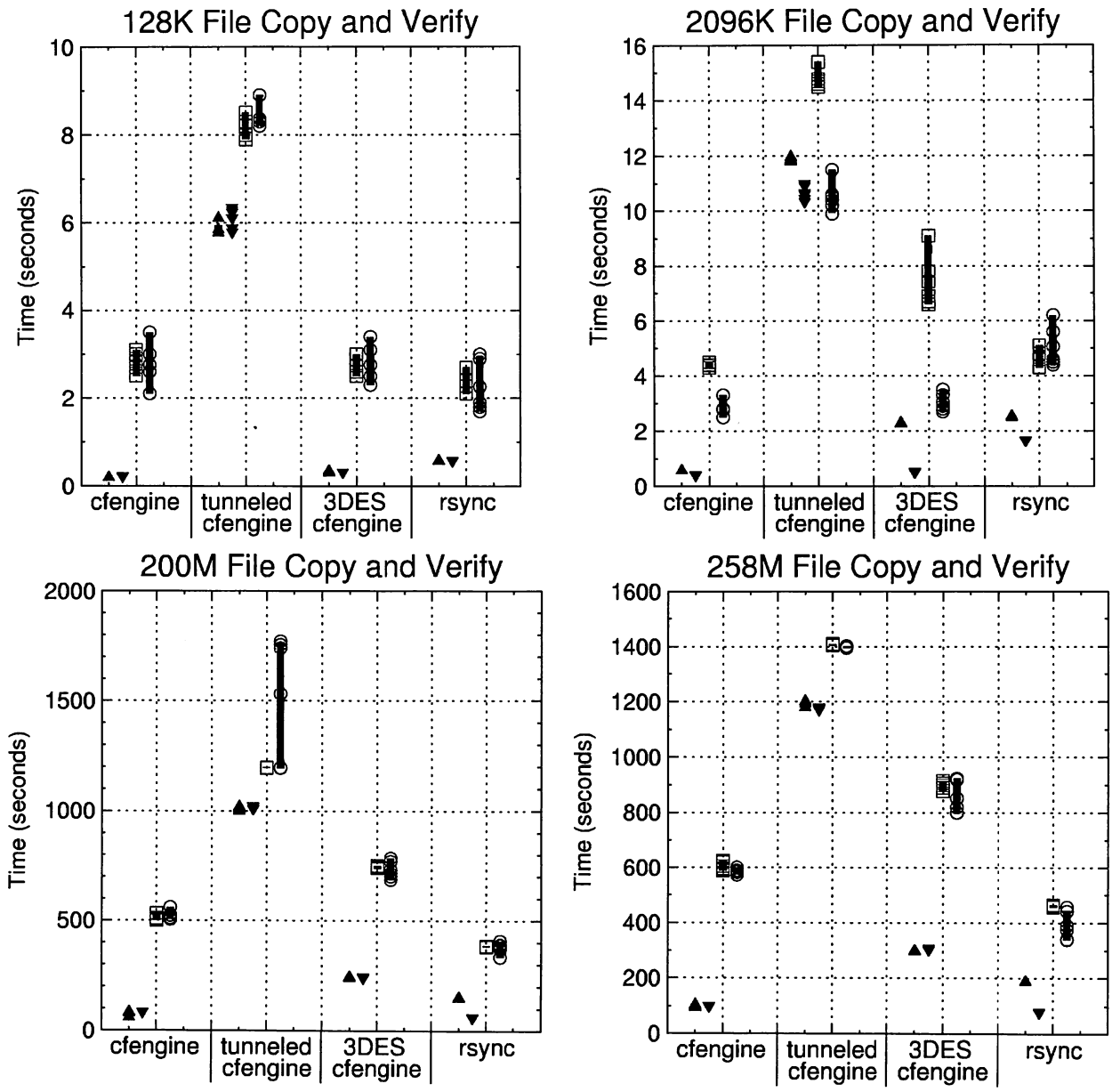


Figure 1: Comparison of file transfer performance.

ran as fast as, or faster than, cfengine on Solaris, under Linux rsync was slower than both cfengine methods until the large transfers, where rsync out-paced 3DES cfengine.

Cfengine tunneled over SSH performed poorly in all tests and showed itself to be a non-valid solution to the problem. This poor performance resulted because communications between cfengine and SSH was through TCP Socket connections. This created more work for the OS's TCP stack since two sockets were opened on each machine (client and server) per connection. In addition to observing a slow initialization and poor general performance, we also found in testing that setting up the tunnel was too brittle to be used in an actual production environment. This instability was attributed to the timings required from opening the tunnel and then initiating the cfengine communication through that tunnel. This timing issue was most problematic when cfengine needed to close and then open a new connect to a cfd server. If the SSH tunnel had not closed before the new cfengine connection was attempted, then a port conflict would arise and the new connection would fail.

What was more interesting was the comparison between rsync and 3DES cfengine, since both used 3DES, from the same OpenSSL [SSL] library, for their cipher. Rsync did suffer in smaller transfers, mostly due to the initialization time required to handle the RSA authentication between hosts. 3DES Cfengine was slower for the large file transfers, mostly due to the extra checks on the client side. The process cfengine used in a transfer was to write the incoming file to a temporary location and then verify that the file was successfully saved before copying it to its final destination on the local disk. It was only a slight increase in I/O utilization in comparison to rsync, but it did slow down the overall transfer process. Those extra checks ensured that the files arrived uncorrupted, which was especially important for distributing critical binaries and data files.

For file verification, the RSA authentication used for rsync created an initial performance hit in opening the connection, which dramatically affected results for smaller file sets. Rsync's checksum algorithm [tridgell] allowed it to best cfengine in the larger file set verifications, even when cfengine was not encrypted. Part of this was because much of the I/O and computational actions are off-loaded by rsync onto the server. With the server being unloaded and having considerably more power than the client, this was a definite plus in our test environment. But this would create scaling problems as demand on the server increased. What constitutes a large demand, though, was not within the scope of our testing. For its file verification, cfengine sent the server the MD5 checksum of its local file. The server then did an MD5 check of its file (which is cached) and responded with the results of the check. While this was a more simplified check than rsync's rolling CRC's, it was just as effective for file verification. The requirement of the client to handle much of the verification load was consistent with the general cfengine design philosophy of "smart client/dumb server" and allowed the server to scale in order to handle more client requests. This did put a slightly greater load on each individual client, although, nothing so significant as to generally hinder the client from performing its primary function.

Conclusions

The results of our tests are preliminary and come from only a single set of parameters. As such, it is difficult to make any definitive conclusions, but we can see trends developing. It can be seen that rsync has advantages over cfengine for large file transport and verification. This is particularly the case where there are a large number of files which have frequent small changes that need to be kept synchronized. On the other hand, if you are interested in synchronizing a few megabytes' worth of less frequently changing data, cfengine is a better choice. This is especially true if you are going to use cfengine's other more interesting features to actually maintain the client hosts. The one advantage that rsync does have over cfengine is the ability to use RSA authentication.

As cfengine evolves, the advantages of rsync for transport and verification of file sets may diminish. The cfengine protocol in the 2.0.x versions is still in the development stages, but already gets rid of the 4096 byte block sizing issue and other transport efficiency issues. Additionally, RSA-style authentication is in the process of being implemented.

Author Information

Andrew Mayhew has a BS-CS from the University of Central Florida in 1997. He stayed in Orlando to work at various ISP's until moving to California to work for Netscape Communication. He left Netscape after the AOL purchase to work as a Senior Internet Infrastructure Engineer for Logictier, Inc. until their closing this September. He continues to live in Sunnyvale, CA and can be reached electronically at amayhew@icewire.com.

References and Further Reading

[cfengine] "Cfengine," http://www.cfengine.org/.

[SSH] "OpenSSH," http://www.openssh.org/.

[SSL] "OpenSSL," http://www.openssl.org/.

[patch] Mayhew, Andrew, "A cfengine Patch Used to Test External Copy Methods," http://icewire.com/cfengine/patches/.

[raw] Mayhew, Andrew, Test scripts, configurations, and results used for this paper, http://icewire.com/cfengine/testing/.

[rsync] "Rsync," http://rsync.samba.org/.

[smith] Smith, Gregory P., "The Perl cfd Replacement Daemon," http://perl-cfd.sourceforge.net/README.perl-cfd.html.

[tridgell] Tridgell, Andrew, and Paul Mackerras, "The rsync algorithm," http://rsync.samba.org/rsync/tech_report/.

CfAdmin: A User Interface for Cfengine

Charles Beadnall – WR Hambrecht
Andrew Mayhew – Logictier, Inc.

ABSTRACT

CfAdmin provides a relatively easy-to-use web interface for system administrators to package and deploy software using GNU Cfengine. It conceals the intricacies of cfengine configuration and reduces potential errors by storing all configuration data in a relational database.

The interface is divided into administration, inventory, package and deployment sections to follow an administrator's work-flow. In addition to implementing most of the features of cfengine, CfAdmin provides a configuration file template system to allow customization of packages per deployment. This allows an administrator to readily leverage a single package, such as Apache, across multiple installations by modifying templated areas of the configuration file (i.e., httpd.conf ServerRoot).

CfAdmin was created to streamline the deployment of software for a managed service provider, but could be equally at home in other environments with large numbers of applications and servers. It is an update to an existing cfengine installation supporting over 200 servers in various colocation facilities and has the side-benefit of being a self-documenting system.

Motivation

As the number of server hosts and server applications we manage has increased during the Internet years it has become increasingly important to efficiently use human resources to manage those hosts. The available workforce continues to be limited, and non-automated software installation processes must be rigorously proceduralized to maintain the documented uniformity required for reliable operation.

Several frameworks have been either developed or glued together to serve the function of centralized server management. PIKT and cfengine are two examples that automated the administration of hosts. While many other software systems are available, each contains the following components, at a minimum:

- Application and configuration repository
- Network distribution system
- Host management/verification agents

Unfortunately, what many of these systems lack is a user interface beyond the configuration files of the management software. These configuration files, in the cases of Cfengine and PIKT, are written in a unique syntax that can take weeks to learn, and a slight error can have disastrous consequences on a network of managed hosts. Training a large group of administrators to maintain and update these configuration files is difficult since most administrators are busy fighting fires and do not have the free time to learn complex and error-prone command structures.

In an attempt to make the underlying management framework easier to use, we developed CfAdmin – a database-backed web interface to manage the management software. The database and interface for CfAdmin glue various administration tools in our framework together and abstract the arcana of proprietary syntaxes.

Interface and Database

Our administration system is composed of the following major parts, with the latter three comprising the CfAdmin interface:

- CVS (v1.10.7) for version control of configuration files
- Cfengine (v1.6.3) server and agent for host management
- Apache Webserver (v1.3.12)
- Tomcat Java servlet engine (v3.2.1)
- Postgres database server (v7.0.1)

The reasoning for our selection of these software packages primarily related to their free availability and previous use experiences. Budgetary considerations also eliminated the few available commercial packages. As an abstracted design, we believe most of these applications could be replaced with functionally similar software. In our design, Apache serves to authenticate users, passing the user name to Tomcat, which performs the bulk of the work using JSPs for formatting and JavaBeans for database access to Postgres.

The database schema was thought out well in advance of much of the interface development and concurrent to the initial deployments of CVS and cfengine. From a design standpoint, it was easier to have all the possible tables and fields we thought we would need in the beginning, rather than trying to add them later to a running production schema. The information we needed to capture fell into four categories:

- **Administration**: Information on users, groups, domains, vendors and authentication rights for CfAdmin.
- **Inventory**: Hardware asset information to correlate machine architecture, operating system, peripheral hardware, and rack/facility location.

- **Package**: Installation specification for individual versions of an application to create distribution packages (similar to an RPM Spec File).
- **Deployment**: Group relationships of hosts to applications, including configuration file customizations.

In a previous prototype, we found administrators would quickly revert back to manual operation when a function was either counterintuitive or unavailable using a graphical interface. To mitigate this risk, we attempted to divide workflows around the common tasks of host inventory management, application package definition, and package deployment.

Administration

In many respects, this portion of the database and interface contains information which, while needed, does not fit into the other categories. This includes information regarding vendor contacts, domains, timezones, countries, and languages. The primary purpose of the administrative section is to deal with the definition of access rights and controls. Group roles are defined and are assigned access rights. Then users are associated with these groups.

In our setup, users are assigned either to facilities, packaging, or operations. People in the facilities group only have access rights to the inventory section of the interface in order to update and maintain facilities-tracking data. Those in the packaging group are the only people allowed to define new packages and to modify existing package definitions. And while operations has read access to all sections, they can only define deployments and make configuration customizations of packages for those deployments. In this way, access controls are maintained.

Inventory

The inventory management section provides a view of systems managed on our network. It captures information related to the location, manufacturer, model, operating system, and network setup of a particular machine. A subset of this hardware information is automatically populated in the database by an audit script when a host is brought up the first time. This automation reduces the amount of time spent on data entry and provides valuable information on the available hardware. CfAdmin provides forms to update this information with anything the audit script missed, including rack elevation and facility location.

Package

The application package section allows the package maintainer to first provide descriptive information about the software package and then to copy the package's files from a CVS repository. This copy process also gathers ancillary information about the file tree, including ownership and permissions. An interface to additional functions such as removing specific files and executing select files on installation is also provided. After the creation of this *gold master* copy, the user can single out configuration files from the imported file system to be modified per deployment. We use a simple templating format to allow the package maintainer to select which areas of the configuration file can be modified per deployment. Beyond these file specifications, the package maintainer can also define actions which need to be done to a package upon deployment. These include making sure certain files contain the proper permissions and ownerships, running pre- and post-install scripts, creating symbolic links so that initialization scripts are run at boot-time, and process checks.

Deployment

The package deployment section provides the ability to create deployment groups, analogous to cfengine Classes, composed of both managed hosts and application packages. The configuration files of packaged applications can also be modified, based on the information provided in the package template.

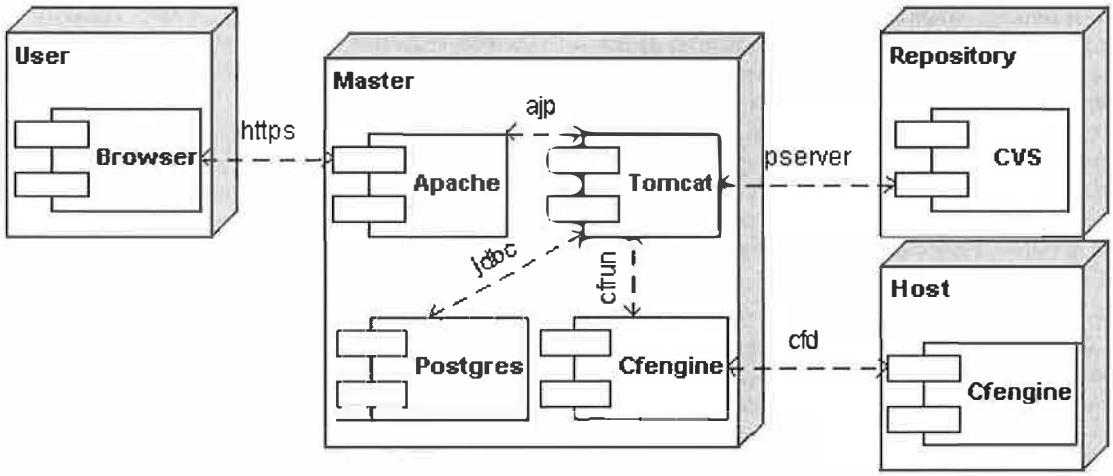


Figure 1: Logical layout of CfAdmin components and communications channels.

In this fashion, an operator can define a deployment group, *Customer 1 Webservers*, which contains hosts *foo* and *bar*. They can then associate the package *Apache 1.3.12 Solaris* to the deployment group. Then they can modify the *httpd.conf* file (based on the configuration template) for the package to listen on port 8080. Once satisfied with the relationship definitions and the configuration customizations, they can just click on the deploy button. This will call a configuration script, which generates the proper cfengine configuration files from the database and then execute the cfengine cfrun agent to force distribution of the application to the newly configured hosts.

Configuration Script

The configuration script's primary function is to act as the glue between CfAdmin and the server-management system. To do this, it takes the information stored in the database regarding packages and deployments, and generates the appropriate configuration files. Our design hope is that analogies between the database schema and the configuration file can be found in any functionally robust server-management system. Admittedly, this is only theory, since we only implemented this glue for one management framework.

For our implementation framework, the script creates cfengine configuration files. From the package section of the database, the script generates a configuration file per package. These configuration files contain the necessary information to install the package files and maintain them. This includes file permission and ownership information along with process-checking and script execution. From the deployment sections of the database, the script creates the classing and control file for cfengine. This file contains all the class definitions and relationships defined by the operations group. In this way, the configuration script is

```
classes:
  midnight = ( Hr00 Min00  )
  Customer1 = ( Customer1_Solaris_webserver  )
  Customer1_Solaris_webserver = ( foo bar  )
  Apache_1_3_12_Solaris_sparc = ( Customer1_Solaris_webserver  )
  Apache_1_3_12_Solaris_sparc_conf = ( Customer1_Solaris_webserver  )

control:
  Customer1_Solaris_webserver::
        Apache_1_3_12_Solaris_sparc_force = ( "true" )
        Apache_1_3_12_Solaris_sparc_class = ( "Customer1_Solaris_webserver" )
        Apache_1_3_12_Solaris_sparc_sched = ( "any" )

import:
  Customer1_Solaris_webserver::  $(CFINPUTS)/packages/cf.Apache_1_3_12_Solaris_sparc
```

Listing 1: A generated cfengine classing and control file.

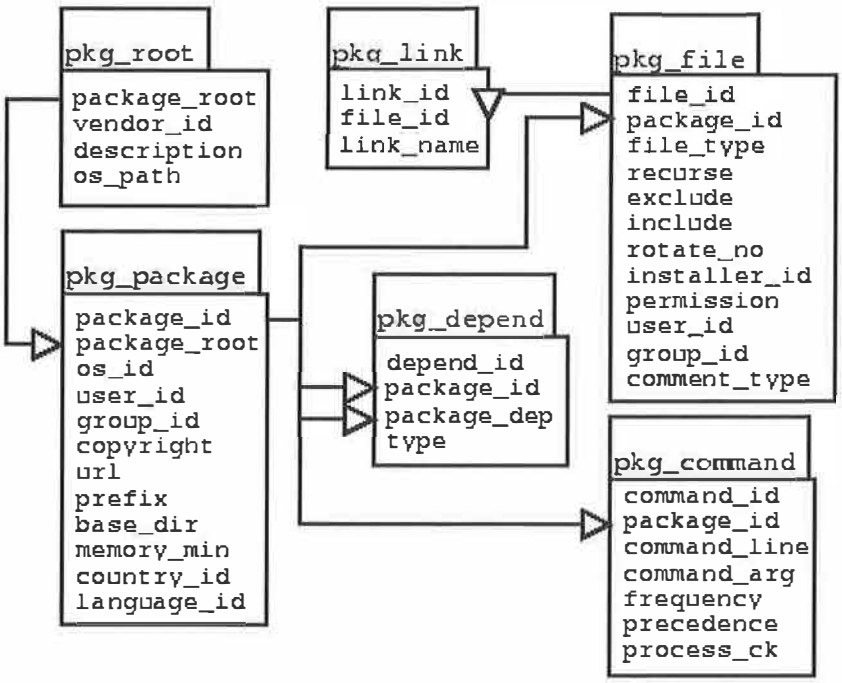


Figure 2: Simplified UML diagram of the packaging section of the database.

able to generate a nearly complete set of cfengine configuration files. The only parts of the cfengine configuration which are not included in the database are implementation-specific bits of data, such as the name of the gold server.

Listing 1 is an example of a generated cfengine classing and control file while Listing 2 shows the cf.Apache_1_3_12_Solaris_sparc package file imported in the previous configuration example.

Future Considerations

In our current implementation, CfAdmin primarily exposes the capabilities of cfengine without expanding on them. A few of the considered expansions include more complete utilization of the CVS repository to provide the ability to tag configurations as "known-good" and to roll back to them in cases of failures. We could also implement an interface to systematically upgrade applications and to schedule deployments at a future date. Scheduled deployments could be useful to plan out long-running maintenance events that span several days. Lastly, CfAdmin by its nature is Unix-centric, and we need to investigate what additional or changed features and functionality the inclusion of Microsoft Windows 2000 management will require.

Author Information

Charles Beadnall is currently responsible for auction operations at investment bank WR Hambrecht in San Francisco. He previously worked for managed service provider Logictier responsible for infrastructure management and for CNN in Atlanta responsible for web site operations. His prior experience includes architecting and implementing Turner Broadcasting's enterprise Internet security systems, managing web development projects at computer manufacturer NCR and managing application support for German Internet service provider Connect GmbH Informations System.

```
#############################################################
## Apache from ASF package classes.
#############################################################
copy:
  Apache_1_3_12_Solaris_sparc::
        /master/package/Apache_1_3_12_Solaris_sparc/apps/apache
                dest=/apps/apache
                server=$(cfd_master)
                recurse=inf
                type=checksum
                force=$(Apache_1_3_12_Solaris_sparc_force)

  Apache_1_3_12_Solaris_sparc_configs::
        /master/configure/$(Apache_1_3_12_Solaris_sparc_class)\
/Apache_1_3_12_Solaris_sparc/apps/conf/httpd.conf
                dest=/apps/conf/httpd.conf
                server=$(cfd_master)
                recurse=inf
                type=checksum
                force=$(Apache_1_3_12_Solaris_sparc_force)

files:
  Apache_1_3_12_Solaris_sparc::
        /etc/init.d/apachectl
                mode=554
                owner=httpd
                group=httpd
                recurse=inf

links:
  Apache_1_3_12_Solaris_sparc::

shellcommands:
Apache_1_3_12_Solaris_sparc.after::
        "apachectl start"

editfiles:
  Apache_1_3_12_Solaris_sparc::
        { /apps/etc/package.lst
                AutoCreate
                SetLine "Apache_1_3_12_Solaris_sparc    $(date)"
                AppendIfNoLineMatching "^Apache_1_3_12_Solaris_sparc.*"
        }
```

Listing 2: The cf.Apache_1_3_12_Solaris_sparc package file.

Andrew Mayhew has a BS-CS from the University of Central Florida in 1997. He stayed in Orlando to work at various ISP's until moving to California to work for Netscape Communication. He left Netscape after the AOL purchase to work as a Senior Internet Infrastructure Engineer for Logictier, Inc. until their closing this September. He continues to live in Sunnyvale, CA and can be reached electronically at amayhew@icewire.com.

References

[apache] "Apache HTTP Server Project," http://httpd.apache.org/.

[burgess] Burgess, Mark, "Cfengine: A Site Configuration Engine," *USENIX Computing Systems*, Vol. 8, No. 3, http://www.iu.hio.no/~mark/papers/cf1.ps, 1995.

[burgess] Burgess, Mark, "Strategies for Distributed Resource Administration Using Cfengine," http://www.iu.hio.no/ ~mark/papers/cf2.ps, *Software-Practice and Experience,* Vol. 27, p. 1083, 1997.

[burgess] Burgess, Mark, *Computer Immunology*, Proceedings: Twelfth Systems Administration Conference (LISA 1998), http://www.iu.hio.no/~mark/research/AIdrift/AIdrift.html, USENIX, 1998.

[cfadmin] "CfAdmin," http://www.icewire.com/cfadmin/.

[cfengine] "GNU Cfengine," http://www.cfengine.org/.

[cvs] "Concurrent Versions Systems," http://www.cyclic.com/cvs/info.html.

[harlander] Harlander, Dr. Magnus, "Central System Administration in a Heterogeneous Unix Environment: GeNUAdmin," *USENIX Proceedings: Eighth Systems Administration Conference (LISA 1994)*, http://www.usenix.org/publications/library/proceedings/lisa94/harlander.html, 1994

[huddleston] Huddleston, Joel, and Steve Traugott, "Bootstrapping an Infrastructure," *USENIX Proceedings: Twelfth Systems Administration Conference (LISA 1998)*, http://www.infrastructures.org/papers/bootstrap/bootstrap.html, 1998.

[jakarta] "Tomcat Java Servlet Engine," http://jakarta.apache.org/.

[pikt] "PIKT, A System Monitor (And Much More)," PIKT's primary task is to warn of problems, but to fix those problems when needed, http://www-gsb.uchicago.edu/pikt/.

[postgres] "PostgreSQL Database Server," http://www.postgresql.org/.

[rpm] "Redhat Package Management," http://www.rpm.org/.

[webmin] "Webmin," http://www.webmin.com/webmin/.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *;login:*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see *http://www.usenix.org/membership/specialdisc.html* for details.

Supporting Members of the USENIX Association:

Addison-Wesley
Kit Cosper
Earthlink Network
Edgix
Interhack Corporation
Interliant
Lessing & Partner
Linux Security, Inc.
Lucent Technologies
Microsoft Research
Motorola Australia Software Centre
New Riders Publishing
Nimrod AS
O'Reilly & Associates Inc.
Raytheon Company
Sams Publishing
The SANS Institute
Sendmail, Inc.
Smart Storage, Inc.
Sun Microsystems, Inc.
Sybase, Inc.
Syntax, Inc.
Taos: The Sys Admin Company
TechTarget.com
UUNET Technologies, Inc.

Supporting Members of SAGE:

Certainty Solutions
Collective Technologies
Electric Lightwave, Inc.
ESM Services, Inc.
Lessing & Partner
Linux Security, Inc.
Mentor Graphics Corp.
Microsoft Research
Motorola Australia Software Centre
New Riders Publishing
O'Reilly & Associates Inc.
Raytheon Company
Remedy Corporation
RIPE NCC
Sams Publishing
SysAdmin Magazine
Taos: The Sys Admin Company
Unix Guru Universe

For more information about membership, conferences, or publications,
see *http://www.usenix.org/*
or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: *office@usenix.org*

ISBN 1-880446-05-7